2017

CALCULATION OF THE EDGE EFFECT OFFSET FOR HIGH EXTRACTION COAL PANELS

Joshua Hescock
University of Kentucky, jrhe227@g.uky.edu
Digital Object Identifier: https://doi.org/10.13023/ETD.2017.347

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Joshua Hescock, Student

Dr. Zach Agioutantis, Major Professor

Dr. Zach Agioutantis, Director of Graduate Studies

CALCULATION OF THE EDGE EFFECT OFFSET FOR HIGH EXTRACTION COAL PANELS

THESIS

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Mining Engineering
in the College of Engineering at the University of Kentucky

By

Joshua Robert Hescock,

Lexington, Kentucky

Director: Dr. Zach Agioutantis, Mining Engineering Foundation Professor

Lexington, Kentucky

2017

Department of Mining Engineering, University of Kentucky, USA

ABSTRACT OF THESIS

CALCULATION OF THE EDGE EFFECT OFFSET FOR HIGH EXTRACTION COAL PANELS

The Surface Deformation Prediction System (SDPS) program has been developed as an engineering tool for the prediction of subsidence deformation indices through the implementation of an influence function. SDPS provides reliable predictions of mining induced surface displacements, strains, and tilt for varying surface topography. One of the key aspects in obtaining reliable ground deformation prediction is the determination of the edge effect offset. The value assigned to the edge effect corresponds to a virtual offsetting of boundary lines delineating the extracted panel to allow for roof cantilevering over the mined out area.

The objective of this thesis is to describe the methods implemented in updating the edge effect offset algorithm within SDPS. Using known geometric equations, the newly developed algorithm provides a more robust calculation of the offset boundary line of the extracted panel for simplistic and complex mining geometries. Assuming that an extracted panel is represented by a closed polyline, the new edge offset algorithm calculates a polyline offset into the extracted panel by the user defined edge effect offset distance. Surface deformations are then calculated using this adjusted panel geometry. The MATLAB® program was utilized for development and testing of the new edge effect offset feature.

KEYWORDS: Edge Effect Offset, Subsidence, Surface Deformation Prediction System, Internal Polygon Offset, High Extraction Coal

Joshua Hescock

August 1, 2017

CALCULATION OF THE EDGE EFFECT OFFSET FOR HIGH EXTRACTION COAL PANELS

By

Joshua Hescock

Zach Agioutantis, Ph.D_____
Director of Thesis

Zach Agioutantis, Ph.D_____
Director of Graduate Studies

8/1/2017_____
Date

Acknowledgements

University of Kentucky College of Public Health

Central Appalachian Regional Education and Research Center (CARERC)

University of Kentucky Mining Department

Zach Agioutantis, Ph.D, Director of Graduate Studies

Table of Contents

List of Tables

List of Figures

1 Introduction

When minerals are excavated from an underground mine that covers a significant area, the overlying rock mass will collapse into the cavities left behind. This process is referred to as subsidence. When subsidence occurs, the formation of hollows, trenches, open cracks, abrupt steps and large troughs are common deformations found on the overlying ground surface. Subsidence causes a vertical displacement of the rock mass and horizontal displacement may often occur as well. Kratzsch, 1983).

The deformation of the ground surface can have adverse effects, e.g., damage to buildings, disruption of communication structures and negative impact on agriculture (Kratzsch, 1983). Additionally, subsidence can impact surface water bodies such as rivers, streams, lakes and wetlands, as well as infrastructure, i.e. roads, railroads, and pipelines (Karmis and Agioutantis, 2015; Newman et al., 2016). Owners of damaged property have taken mining operations to court in order to receive compensation for damages (Kratzsch, 1983) and the Environmental Protection Agency (EPA) and Army Corps of Engineers regulate the effects of mining on the local environment.

The original purpose of subsidence prediction was to determine the extent of surface damage, the duration of surface influence from mine workings, and to attribute the appropriate share of costs. More in-depth procedures in subsidence estimation have become necessary due to the increase in underground mining, along with the increase in surface residential development. Today it is necessary for subsidence estimation to predict strata and ground movements over mine operations in order to determine the influence of such movements on building structures, mine shafts and other surface structures. These estimations also help to minimize subsidence damage by means of improvements in mining, protection of structures and regional planning (Kratzsch, 1983).

Subsidence prediction is a necessary part of any mine operation. Estimations for subsidence prediction are necessary in order to obtain mining permits before any excavation can begin. Government organizations such as the EPA and Army Corps of Engineers require subsidence estimation to be performed in order to understand what impact underground mining operations will have on the local hydrology and infrastructure such as roads, railroads or buildings. Accurate subsidence prediction is important in order for the mining company to avoid citations for damage to the local environment or hydrology and to avoid lawsuits which would require the mine to compensate the infrastructure owners for any damages that are a direct result of the mining excavation.

This thesis seeks to create a polygon offsetting algorithm to determine the inward offset polygon coordinate locations given the coordinates of an input polygon and an offset distance. The purpose of creating this algorithm is to improve the estimation of surface subsidence above underground longwall panels and to update the algorithm within the

Surface Deformation Prediction System (SDPS) with the developed algorithm. The calculation of the inward offset polygon is necessary in the estimation of surface subsidence and by improving the inward offsetting algorithm there will also be an improvement in the estimation of the surface subsidence.

This thesis will discuss surface subsidence, including its importance, and give a short discussion of the different methods that are available to estimate its magnitude. The algorithm will use the Influence Function method and a detailed discussion will be given to explain how this method estimates surface subsidence including a description of the parameters used. A description of the SDPS package will follow explaining how it estimates surface subsidence and the methods that it uses to perform the calculations.

A discussion of the importance of the edge effect offset in the estimation of surface subsidence will be included and followed by a short discussion of other existing methods that perform the inward offsetting of polygons. Examples are included in this section of the AutoCAD offsetting function and the inaccuracies that were encountered when using its offsetting function as the standard against the new algorithm.

An in-depth description of the developed algorithm will follow including case studies. The case studies evaluate the ability of the algorithm to correctly determine the inward offset polygon and will contain figures that compare the results of the algorithm's methodology and the offsetting function within AutoCAD. In addition conclusions about the results of the new algorithm will be discussed.

2 Surface Subsidence Prediction

There are various methods with which to estimate surface subsidence above an extraction panel.

The Profile Function Method defines the distribution of subsidence values on the surface along an axis orthogonal to the boundary of a theoretical, infinitely long, underground excavation. The parameters for the Profile Function Method must be determined from field data. There are multiple prediction models available in the literature that can be used to determine the subsidence profile for a given coalfield. A prediction model used for this method applicable for the Eastern US coal fields is based on the hyperbolic tangent formulation shown below (Agioutantis and Karmis 2016).

$$S(x) = \frac{1}{2} S_{max}\{1 - \tanh\left[\frac{cx}{B}\right]\} \quad (2.1)$$

Where:

S(x) = subsidence at x
x = the distance from the inflection point
Smax = the maximum subsidence of the profile
B = the distance from the inflection point to point of Smax
c = a constant.

This method is sensitive to the maximum subsidence factor, *Smax*, as well as the distance of the inflection point from the rib.

The zone area method was developed by J.E. Marr as an adaptation of the Influence Function Method. This method, also known as the circle method, predicts surface subsidence by constructing a number of concentric zones around a surface point, the radius of the outer zone being equal to the radius of the area of influence. The subsidence at the surface point is calculated by summing up the proportions of coal that are extracted within each of the zones and then multiplying this value by the subsidence factor. This method allows the estimation of subsidence even for extraction zones of irregular shape (Bell, 2013).

An example of the zone area method is shown in Figure 1. The width of each zone is equal to one-tenth of the depth of the extraction zone.

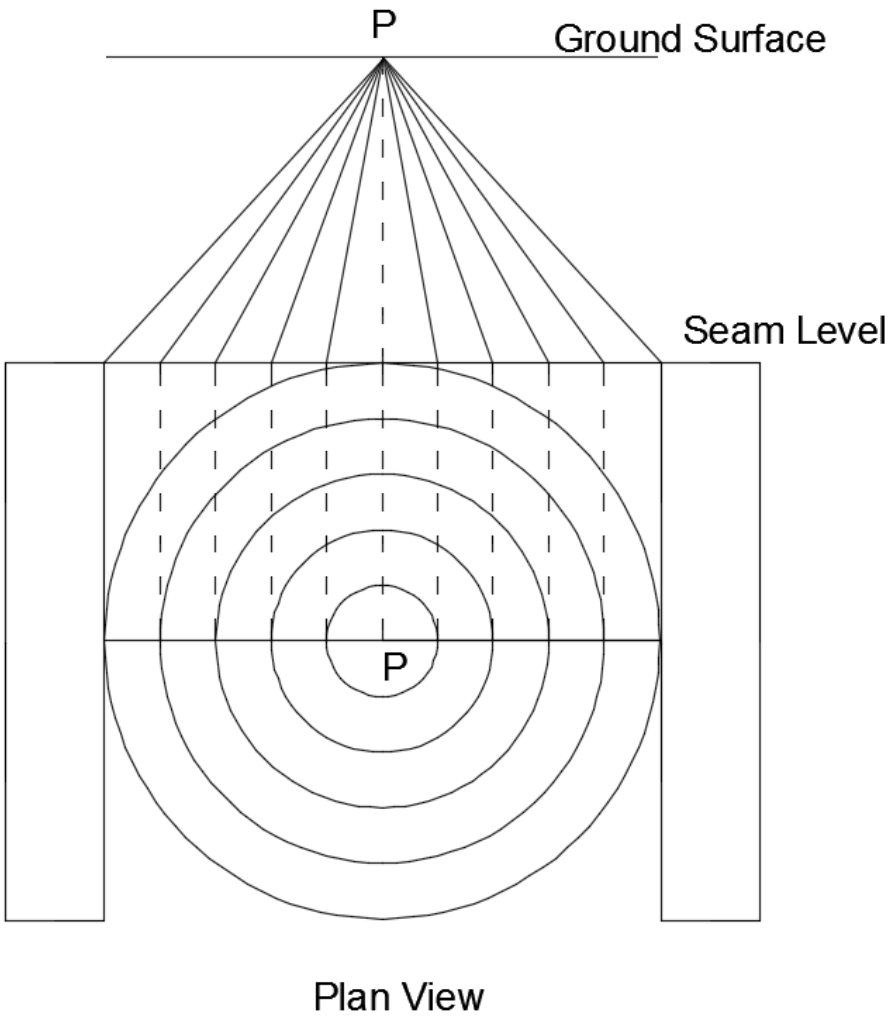

Figure 1: Zone Area Method of surface subsidence prediction.

2.1 Review of Methods

Underground operations have been known to cause mining-induced subsidence inside and outside of the mine permit area (Karmis, Agioutantis, and Andrews, 2008). Additionally, subsidence can impact surface water bodies, such as rivers, streams, lakes, and wetlands (Karmis and Agioutantis, 2015; Newman et al., 2016), as well as infrastructure, such as roads, railroads, pipelines, and buildings. With an increase in regulatory focus on mining-induced impacts to local structures, the calculation of mining-induced subsidence indices has become necessary in determining potential damage and mitigation efforts. The accurate prediction and evaluation of mining-induced ground deformations is required to obtain underground mining permits due to potential impacts to nearby structures, as well as hydrogeological sources. The prediction of ground deformation indices is often a complex task due to the number of input parameters

required, such as mining geometries, overburden lithology, surface topography, and the rate of mining. (Karmis, Agioutantis, and Andrews, 2008).

2.2 The Influence Function Method

The influence function, as implemented by SDPS, is the bell-shaped Gaussian function often also referred to as Knothe's Prediction Method (Knothe, 1957).This method assumes that the influence function for the two-dimensional case is given by Equation 1.1.

$$g(x,s) = \frac{S_o(x)}{r} \exp[-\pi \frac{(x-s)^2}{r^2}] \tag{2.1}$$

Where:

r = the radius of principal influence = h / tan(β)
h = the overburden depth
β = the angle of principal influence
s = the coordinate of the point, P(s), where subsidence is considered
x = the coordinate of the infinitesimal excavated element
So(x) = the convergence of the roof (subsidence factor) of the infinitesimal excavated element

Subsidence at any point P(s) can be expressed by Equation 1.2.

$$S(x,s) = \frac{1}{r} \int_{-\infty}^{+\infty} S_o(x) \exp[-\pi \frac{(x-s)^2}{r^2}]\ dx \tag{2.2}$$

Where:

So(x) = Smax = m(x)a(x)
m(x) = the extraction thickness
a(x) = the roof convergence or subsidence factor

Alternatively, Equation 2 can be rewritten to calculate subsidence at any point P(s) within the boundary limits of x_1 and x_2.

$$S(x,s) = \frac{S_{max}}{r} \int_{x_1}^{x_2} \exp[-\pi \frac{(x-s)^2}{r^2}]\ dx \tag{2.3}$$

When taking the edge effect offset, d, into account the previous equation then becomes:

$$S(x,s) = \frac{S_{max}}{r} \int_{x_1+d}^{x_{2-d}} \exp[-\pi \frac{(x-s)^2}{r^2}]\ dx \qquad (2.4)$$

The edge effect offset is the distance of the inflection point of the subsidence profile from the rib of the excavation. The magnitude of the offset is related to the properties of the rib, specifically how much the rib yields or how resistant the rib is to yielding. This offset distance also indicates the location of the inflection point, which designates the transition from horizontal tensile zones to compressive strain zones.

2.3 Parameters used by the Influence Function

2.3.1 Supercritical Subsidence Factor

Also known as Smax, it is the maximum expected subsidence within the subsidence trough. The inflection point can be found a distance of Smax/2 away from the ribs of the excavation towards the panel center.

2.3.2 Angle of Principal Influence

The angle of principal influence (β, beta) or the angle of influence is one of the basic parameters used in the influence function method since it has a major impact on the distribution of the deformations on the surface. It is the angle between the horizontal and the line connecting the projection of the inflection point position of the subsidence trough, at the seam level, with the surface point of "zero influence", i.e., where subsidence is about 0.6 percent of the maximum subsidence value (VPI & SU 1987). The average value determined for the Appalachian coalfields is $\beta \simeq 67$ deg. The parameter required for these calculations is the tangent of this angle (e.g., tanβ = 2.31). The angle of influence is related to the radius of influence as shown in the equation:

$$\tan\beta = \frac{h}{r} \qquad (2.5)$$

Where

h = the overburden depth
r = the radius of influence.

This value should be determined for each site by fitting a calculated subsidence profile to a measured subsidence profile. If this is not possible, the influence angle can be approximately set as the complementary angle to the angle of draw.

2.3.3 Edge Effect Offset

The edge effect offset, *d*, is the distance of the location of the inflection point of the subsidence profile from the rib of the excavation. The magnitude of the edge effect offset is mainly related to the amount of yielding of the rib due to panel advance or panel mining. Also, the edge effect may be impacted by the rigidity of the immediate roof strata.

The estimation of the edge effect offset can be determined from the relationship between the depth of the mine panel and the largest width across the panel. Figure 2 shows the chart that estimates the edge effect offset using the width-to-depth ratio.

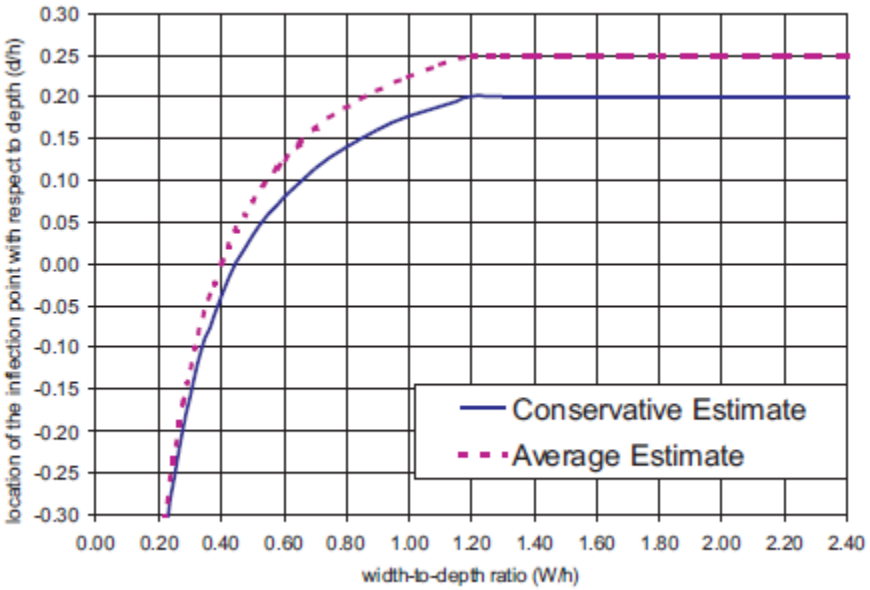

Figure 2: Width-to-Depth Ratio chart to estimate the edge effect offset distance (Agioutantis and Karmis, 2016).

Typical parameters needed for the implementation of this method within SDPS are the angle of influence, the supercritical subsidence factor, and the edge effect offset distance. The edge effect offset distance is estimated using empirical relationships as a function of the width-to-depth ratio. The supercritical subsidence factor can be calculated as a function of the percent of hard rock present within the overburden (Agioutantis and Karmis 2016). The ribs of the excavated panel are adjusted inward by the offset distance.

Figure 3 shows an example of a simple, rectangular mine plan that has been offset inwards by a distance equal to a user-defined edge effect offset. Line A-A' is a cross-section along which subsidence and strains will be calculated. Point B is the location of the rib on the original mine plan and Point C is the respective location on the edge effect offset panel.

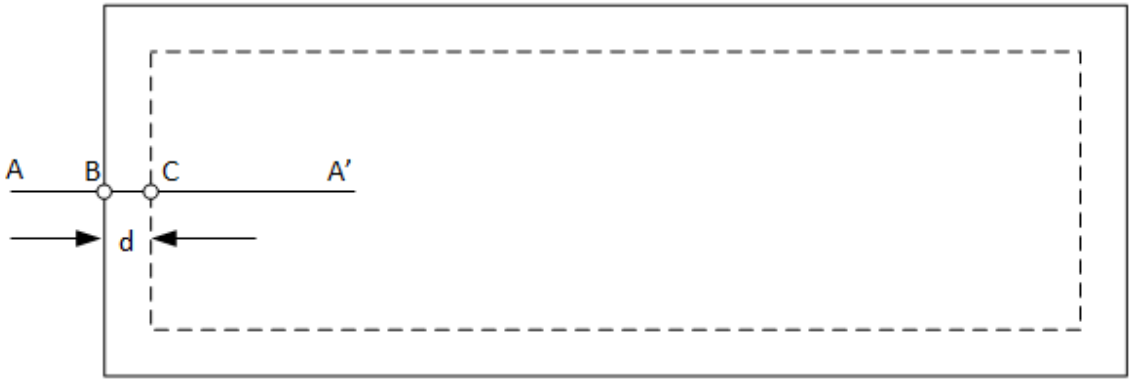

Figure 3: Adjustment of a rectangular panel's boundaries by the edge effect offset, d.

Figure 4 shows an example of a polygonal mine plan that has been offset inwards by a distance equal to a user-defined edge effect offset. Line A-A' is a cross-section along which subsidence and strains will be calculated. Point B is the location of the rib on the original mine plan and Point C is the respective location on the edge effect offset panel.

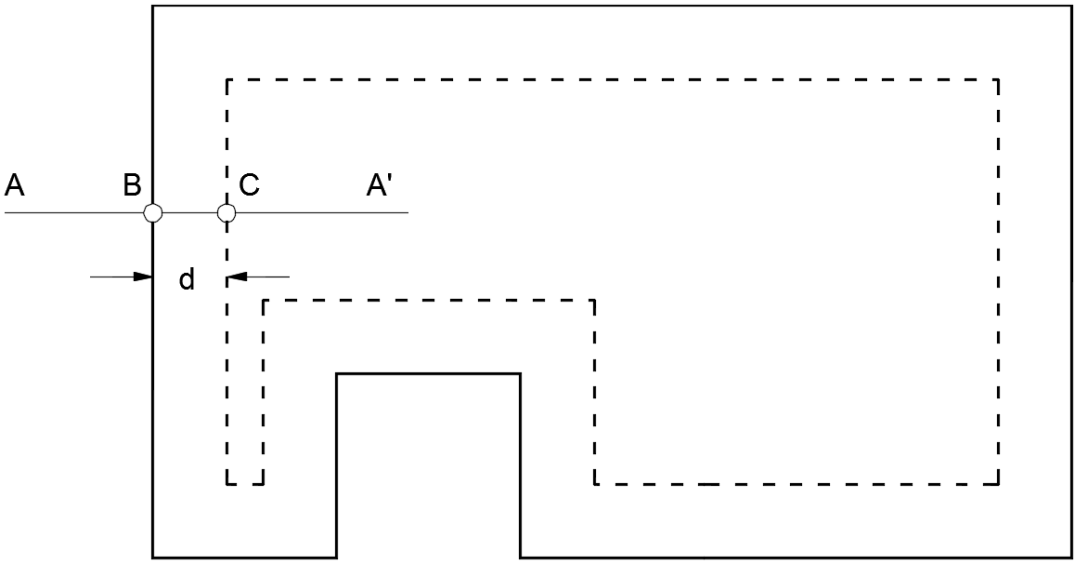

Figure 4: Adjustment of a polygonal panel boundary by the edge effect offset, d.

Figure 5 shows how the influence functions are shifted from the original panel boundary to the adjusted panel boundary. By shifting the influence of each unit element excavated at seam level, the overall panel influence area changes. Subsidence calculated using the unit element at Point B is defined by the thin line on the left. The edge effect offset causes the unit element at Point C to be used instead and results in the subsidence represented by the thicker line on the right. This example is correct when referring to either a rectangular mine plan, Figure 3, or a polygonal mine plan, Figure 4.

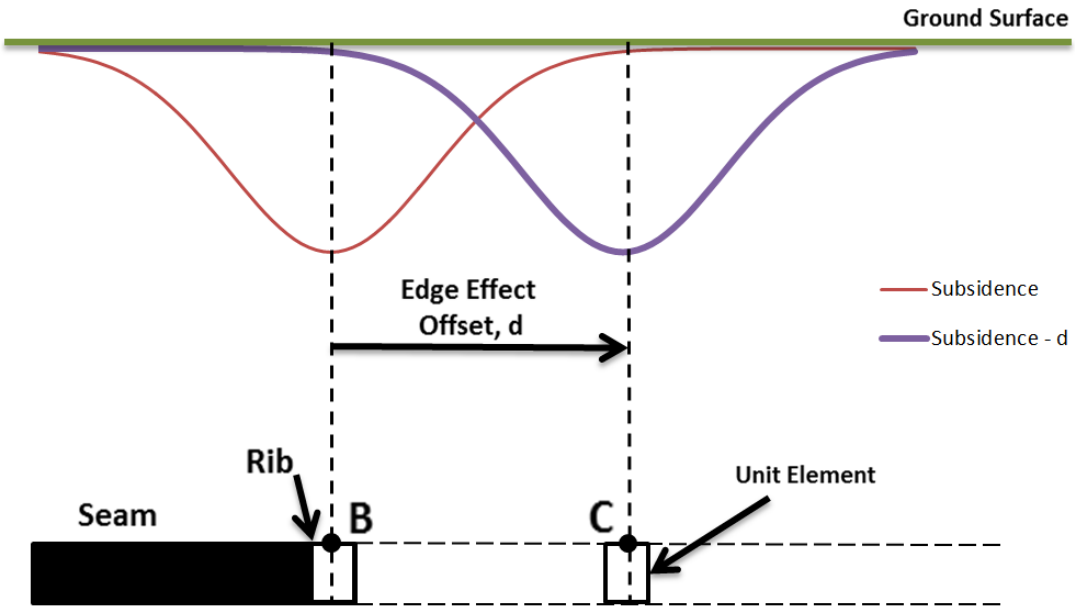

Figure 5: Change in subsidence calculations due to the edge effect offset.

2.3.4 Overburden Depth

The overburden depth is the vertical distance between the ground's surface and the roof of the mine panel being excavated. Overburden depth is reported in feet or meters.

2.3.5 Extraction Thickness

The average vertical distance between the excavated mine panel's roof and floor. Extraction thickness is also referred to as the average height of material to be extracted within the panel.

2.3.6 Percent Hardrock

The percent "hardrock," as defined in subsidence investigations, represents the sum of the strong rocks (e.g., sandstone, limestone), having a minimum thickness of 5 feet, expressed as a percentage of the total overburden thickness.

2.4 The SDPS Package

The Surface Deformation Prediction System (SDPS) was developed by the Department of Mining and Minerals Engineering, Virginia Polytechnic Institute and State University (Virginia Tech). The SDPS package is used to calculate various ground deformation parameters through the implementation of empirical relationships that have been validated through numerous case studies (VPI&SU, 1987). SDPS is used widely in academia, industry, and regulatory agencies to predict mining-induced subsidence and evaluation of mitigation efforts.

SDPS uses two different methods for calculating ground deformations at user-defined evaluation points caused by high extraction underground mining: the profile function method and the influence function method (Newman, Agioutantis, and Karmis, 2001). The influence function method is a mature methodology for the calculation of ground deformations with respect to high extraction mining operations. These calculations can be performed with respect to empirically derived or observed specific parameters providing parameter flexibilities while maintaining a dependable level of accuracy (Agioutantis and Karmis, 2013).

The estimation of subsidence is performed by utilizing a user-input mine plan and user-input prediction points—along which subsidence parameters will be calculated. SDPS can handle two types of mine plans: polygonal or rectangular. A rectangular mine plan consists of a simple rectangular-shaped mine plan. A polygonal mine plan is more complex as the mine plan contains an irregular shape due to the presence of pillars or cutouts in the ribs. This type of mine plan is harder to estimate subsidence due to its complex shape.

Prediction points can also be classified into two categories: line or grid. A line of prediction points contains a series of coordinates limited to one dimension, a cross-section, whose locations are chosen by the user in order to extend beyond the ribs of the mine plan and the number of prediction points is chosen to increase visual clarity. A grid of prediction points has the same properties as a line of prediction points; however a grid extends in two dimensions, calculating subsidence parameters along multiple cross-sections in series for the user to analyze. Prediction points are used to calculate the amount of subsidence at and around each prediction point and SDPS can graphically show the magnitudes of a chosen subsidence parameter along one or multiple cross-sections defined by the prediction points.

3 Application of the Edge Effect Offset

3.1 Introduction

The estimation of subsidence uses a series of calculations using parameters taken from the local geology in order to estimate the effects of subsidence on the ground surface as accurately as possible. If any of the parameters used within the calculations are inaccurate, then the amount of estimated subsidence will also be inaccurate. It is important to maintain the highest level of accuracy possible when determining the edge effect offset distance.

The edge effect offset is mainly related to the amount of yielding of the ribs due to panel advance or panel mining and may be impacted by the rigidity of the immediate roof strata. As shown in Figure 5, the edge effect offset can shift the location of the subsidence trough. One major priority during in the planning and development of a mine is to avoid potential damage to infrastructure and local hydrology as even a subtle shift in the location of the subsidence trough could result in damage to these structures.

3.2 Method of Offset Drawing

One of the major steps within the calculation of surface subsidence is the application of the edge effect offset distance onto the extraction panel. In situations where the edge effect offset distance is a value of zero, the locations of the ribs on the extraction panel will be used in the estimation of subsidence. If the value of the edge effect offset distance is non-zero, then the estimation of subsidence requires a new polygon on which to perform the calculations. This new polygon will be offset inwards from the extraction panel by a distance equal to the edge effect offset distance which was previously calculated based on the amount of yielding expected from the ribs of the extraction panel.

Literature review has shown that there are different methods for determining the inward offset polygon as well as the different applications with which this process can be used.

Xiaorui Chen and Sara McMains introduce a new algorithm that they developed in order to perform the inward offsetting of polygons by using winding numbers in order to improve machining by determining the accessible area for a given tool radius. The first step offsets the edges inward along the inward edge normal direction by the offset distance chosen, Figure 6(a). If the vertex is concave, defined as a vertex which takes a right turn when moving along the contour, then the endpoints of the offset edges that originally shared the vertex are connected by a concave arc whose center is on the original vertex. If the vertex is convex, defined as taking a left turn when moving along the contour, then the endpoints are connected with straight lines that also connect to the original vertex, Figure 6(b).

The winding numbers are then calculated for each region within the raw offset curve. Regions that have a positive winding number are considered inside the inner offset polygon, Figure 6(c). The boundary of the union of all regions with a positive winding number is determined to be the final inner offset polygon, Figure 6(d) (Chen and McMains, 2005).

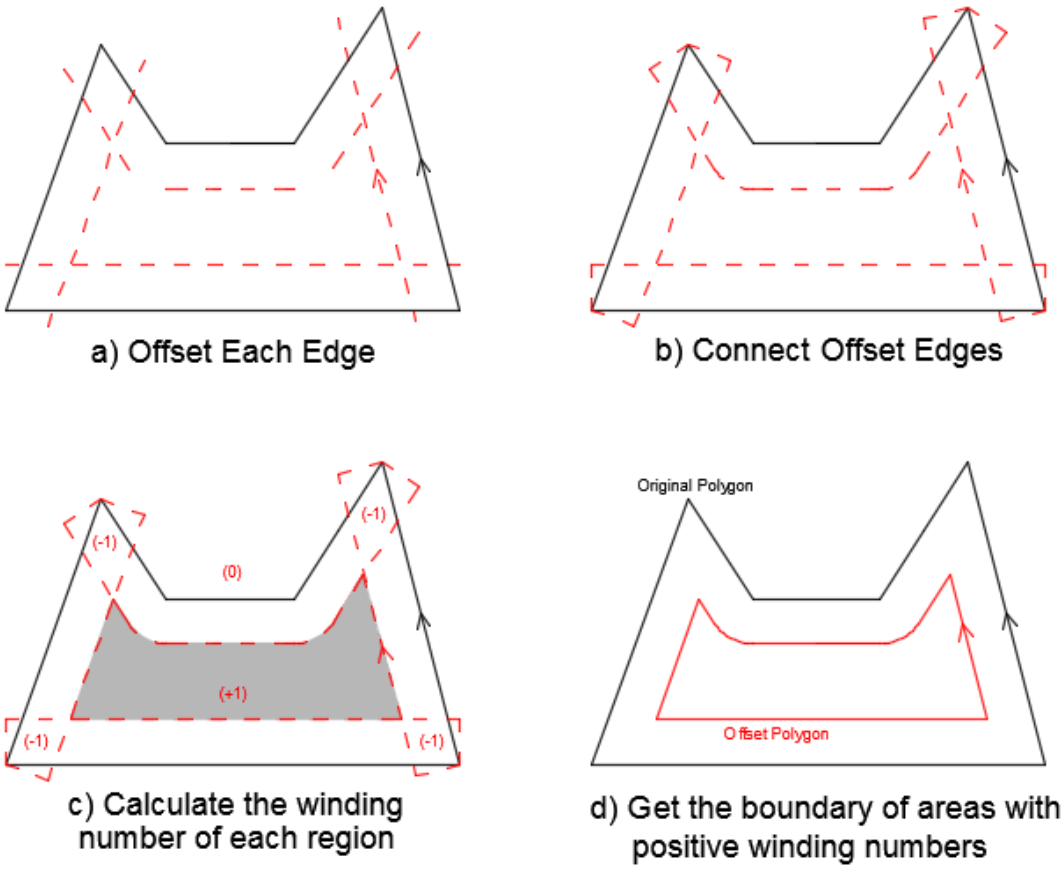

Figure 6: Construction of an inward edge offset polygon.

3.3 Limitations within the AutoCAD Method

During the creation of the new offset algorithm, tests were performed to examine the accuracy of the algorithm by using the offsetting function within AutoCAD as the standard. During these tests, it was discovered that under certain conditions the offsetting function within AutoCAD was producing results that were inappropriate for subsidence prediction.

Without an in-depth description of the methodology used within the AutoCAD offsetting function, it cannot be determined whether these limitations are the result of a design error within the AutoCAD offsetting function or if the example polygons used were working outside of unknown constraints defined within the AutoCAD function. Regardless of the

source of these limitations, the AutoCAD offsetting function will be used as the standard to assess the accuracy of the developed algorithm.

The polygon used when the limitation was discovered is show in green in Figure 7. It was used as an accuracy test due to the large occurrence of pillars that protrude into the panel. The large number of pillars was used to test the updated algorithm on its ability to handle the negative spaces that are created when offset line segments overlap each other.

An edge effect offset distance of 30 feet was chosen to test how AutoCAD and the new algorithm were able to calculate the final inward offset polygon. The results of the offsetting function within AutoCAD are shown in Figure 7 using the default settings that are used within AutoCAD 2015.

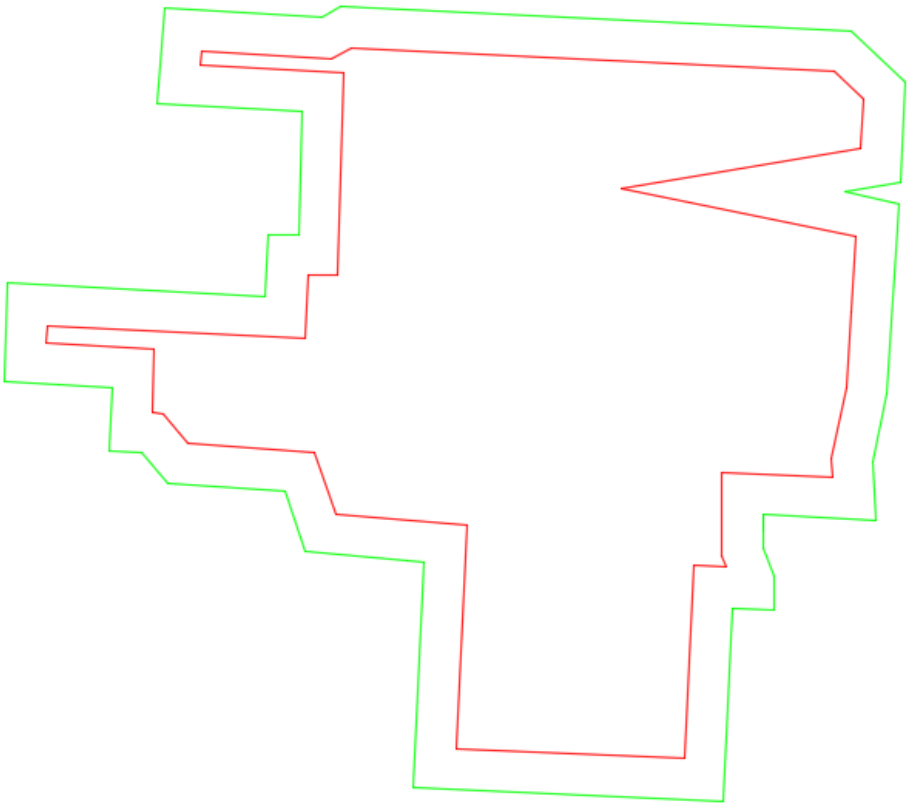

Figure 7: AutoCAD output for an original mine panel with offset polygon for a 30 foot edge effect offset distance.

The original polygon is shown in green and the inward offset polygon is shown in red. The inward offset polygon follows the shape of the original polygon very well, including the narrow openings in the northwest and western edges. The limitation mentioned previously can be found on the eastern edge. The small triangular point protruding into

the original polygon results in a very large triangular point being formed within the inward offset polygon.

When considering the small size of the original triangle in relation to the total size of the polygon, it is shown that at a large enough edge effect offset the offsetting function within AutoCAD will extend the tip of the triangle until it reaches the opposite side of the inward offset polygon. To test how AutoCAD handles this type of situation an edge effect offset of 65 feet was chosen as shown in Figure 8.

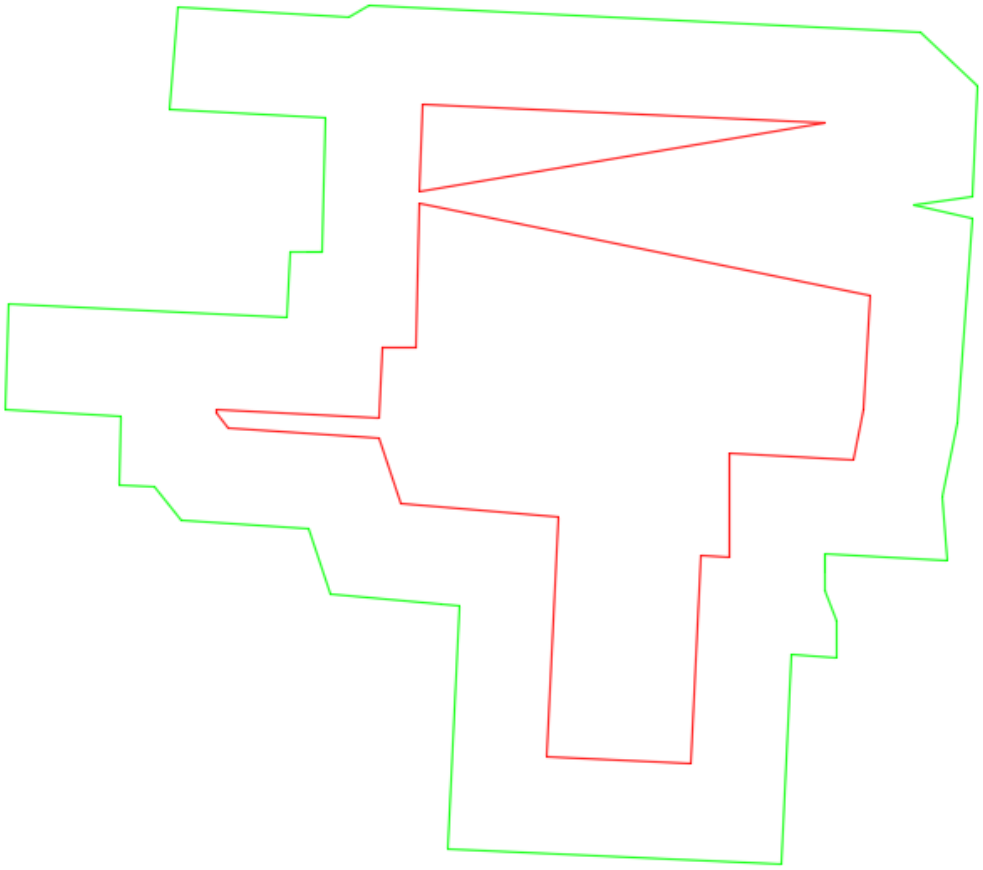

Figure 8: AutoCAD output for an original mine panel with offset polygon for a 65 foot edge effect offset distance.

Figure 8 reveals that the default AutoCAD offsetting function will continue to extend the tip of the triangle until it encounters the opposite edge of the inward offset polygon. As a result, any edge effect offset distance that is too large will result in the inward offset polygon being split into two final offset polygons as seen in Figure 8. The area located between the two offset polygons is considered to not contribute in the final offset polygon. If SDPS used this inward offset to estimate surface subsidence, the estimated

subsidence might be significantly inaccurate and could result in damage to infrastructure, local hydrology and miners.

To compare the results produced with AutoCAD with the new algorithm, the same polygon coordinates were entered into the algorithm and the inward offset polygon was determined for an edge effect offset distance of 30 feet, Figure 9, and 65 feet, Figure 10.

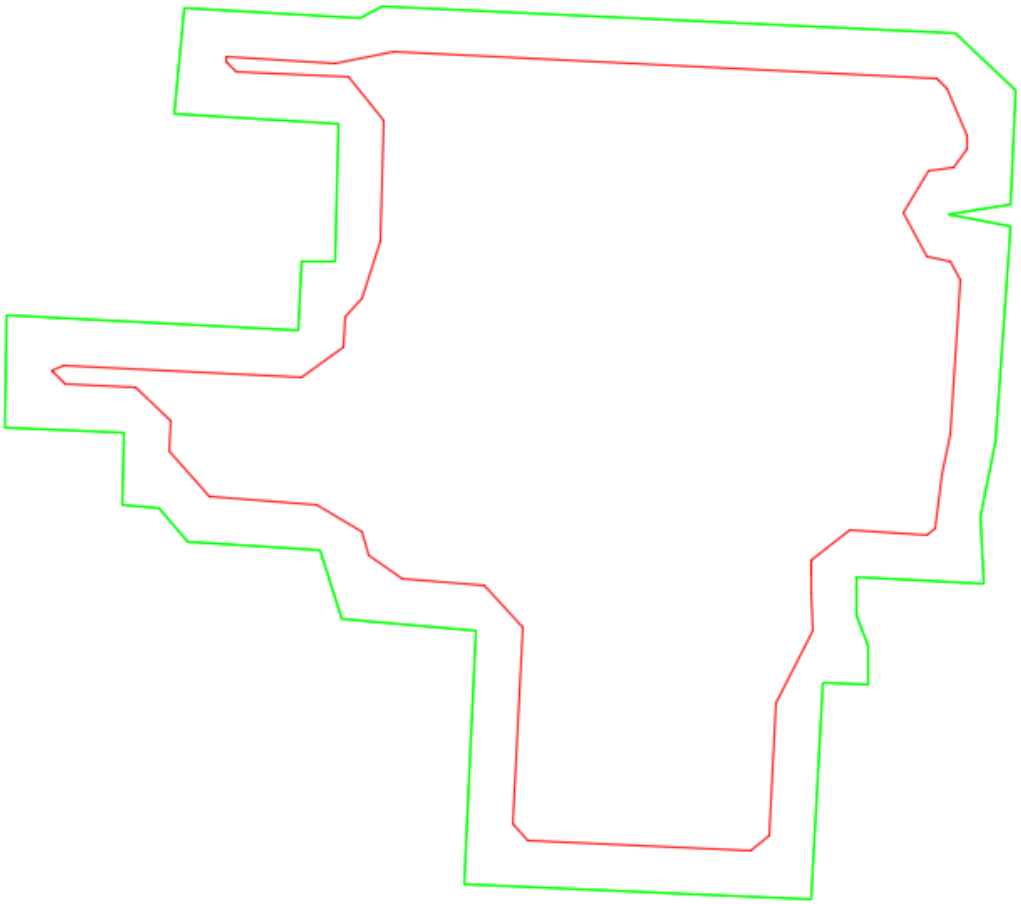

Figure 9: Algorithm output for an original mine panel with offset polygon for a 30 foot edge effect offset distance.

Figure 9 shows the 30 foot inward offset polygon for the same original polygon as in the AutoCAD examples. The coordinates were recorded from Matlab and transferred into AutoCAD to maintain a constant figure format.

The original polygon is shown in green and the inward offset polygon is shown in red. A noticeable difference between the AutoCAD outputs and the algorithm outputs is the more rounded corners displayed in the algorithm outputs. The possible causes for the issue with the triangle which split the inward offset polygon into two polygons in AutoCAD were examined and the algorithm was designed to combat this issue. During

the calculation of the inward offset polygon, the algorithm recognizes the relationship between two edges as they meet and will remove the tip of a corner when an arrowhead shape is pointing into the original polygon at sharp enough angles. A more in depth description about this method can be found in section 3.5.9. In addition a chamfer function was added to the last steps of the new algorithm which results in all locations on the final offset polygon that form an arrowhead which points towards the center to have the point removed.

Despite the differences in the steps used to calculate the inward offset polygon, the results in Figure 9 reveal that the algorithm is capable of producing an accurate representation of the inward offset polygon with minimal differences compared to the AutoCAD results.

Figure 10 reveals the output of the algorithm for an edge effect offset distance of 65 feet. The differences between the AutoCAD and the algorithm outputs become much easier to notice. As a result of the design within the algorithm to remove the tips of arrowheads that have sharp enough angles within the original polygon, the error found in the AutoCAD offsetting function, Figure 8, is not present within the algorithm's inward offset output. The tip of the arrowhead was removed during the offset calculations resulting in a flat edge instead of a point.

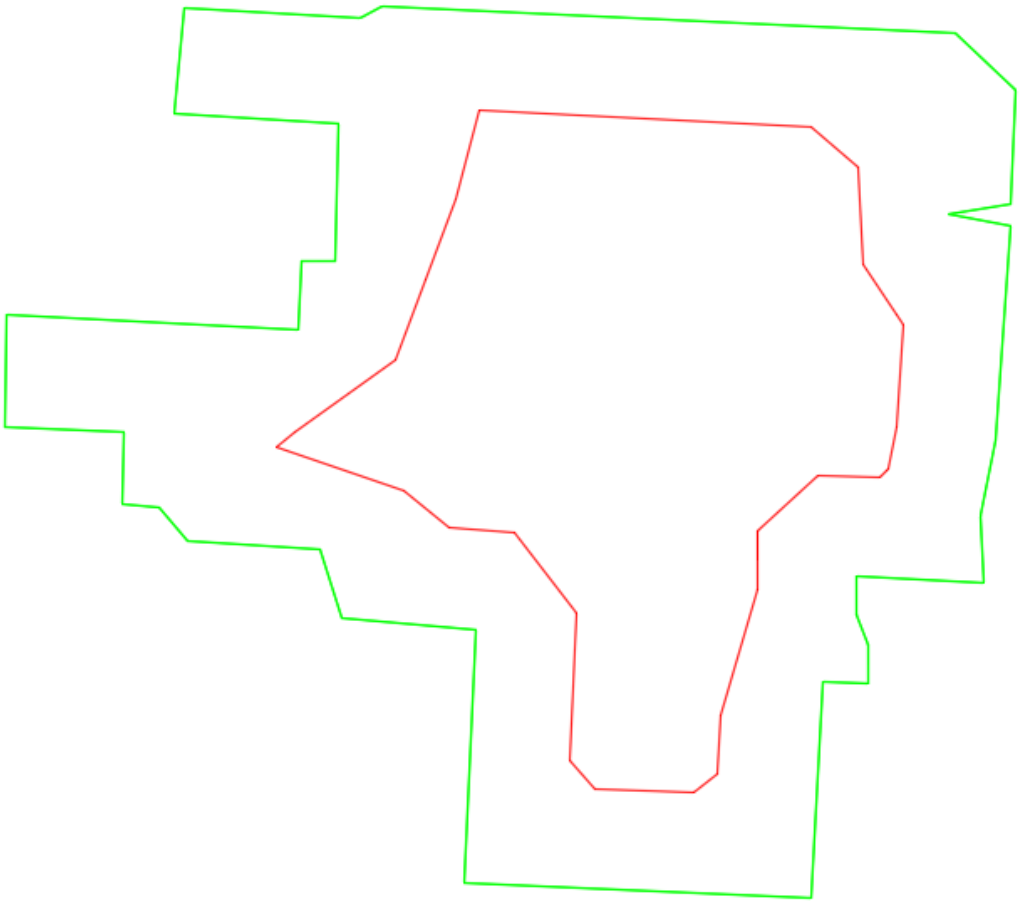

Figure 10: Algorithm output for an original mine panel with offset polygon for a 65 foot edge effect offset distance.

Another difference between the two offsetting functions is the smoother shape that the new algorithm creates while the results from the AutoCAD are more angular and sharp. This is a side-effect of the additional chamfer function added to the final steps of the new algorithm. The offset polygon in Figure 8 has a small, sharp section that enters the narrow opening at the western edge while the offset polygon in Figure 10 has a triangular point that protrudes into the narrow opening at the western edge.

3.4 Development of an Advanced Algorithm for Underground Mine Panels

Without knowing the methodology used in the AutoCAD offsetting function, it became necessary to develop a new offset function that would not produce offset polygons that contain the same limitations as the AutoCAD or the SDPS offsetting functions. Many of the subroutines used within the algorithm are developed from geometric principles. A flowchart of the steps used within the algorithm can be found in Figure 12Figure 12.

Before examining the purpose of each individual step within the algorithm, it is important to note that the methodology used in the developed algorithm creates corners by extending the endpoints of the offset edges until they intersect. This results in square or angled corners, as shown in Figure 11.

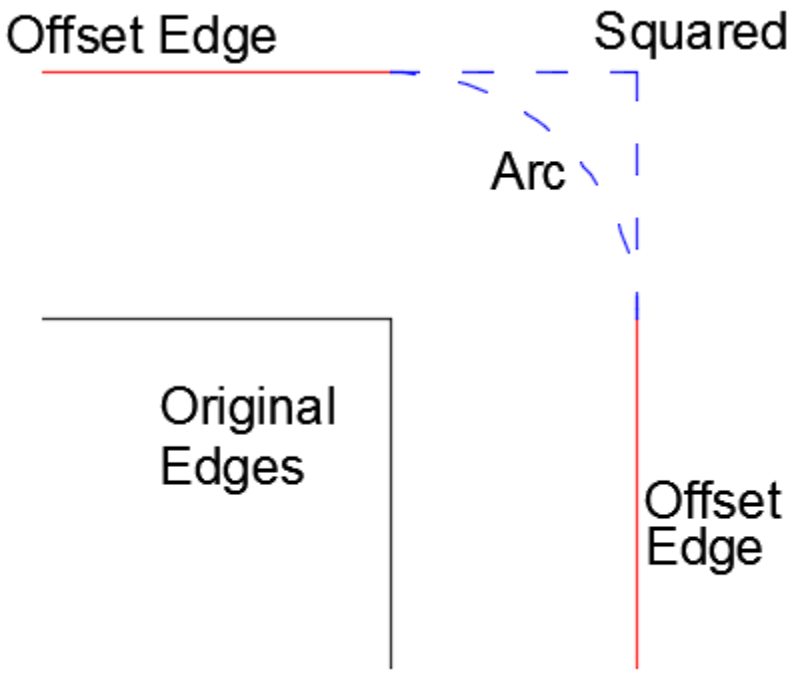

Figure 11: An example of using arcs or angles to connect offset endpoints.

The more appropriate method would connect the endpoints of offset edges using arcs because the distance from every point within the arc to the vertex shared by the two original edges is equal to the edge effect offset distance. When using a squared corner, the point located on the corner, the farthest point away, is located at a distance greater than the edge effect offset. However, the percent difference in total area between the two methods can be considered negligible.

The use of square or angled corners was chosen due to the increased difficulty needed in order to create arcs because an arc is made of many very small segments. In addition, the large number of line segments contained within each of the arcs created would greatly increase the amount of time that the algorithm would take to perform an inward offset. By using angled corners, the runtime of the algorithm is more efficient and it was not necessary to develop a function to create arcs.

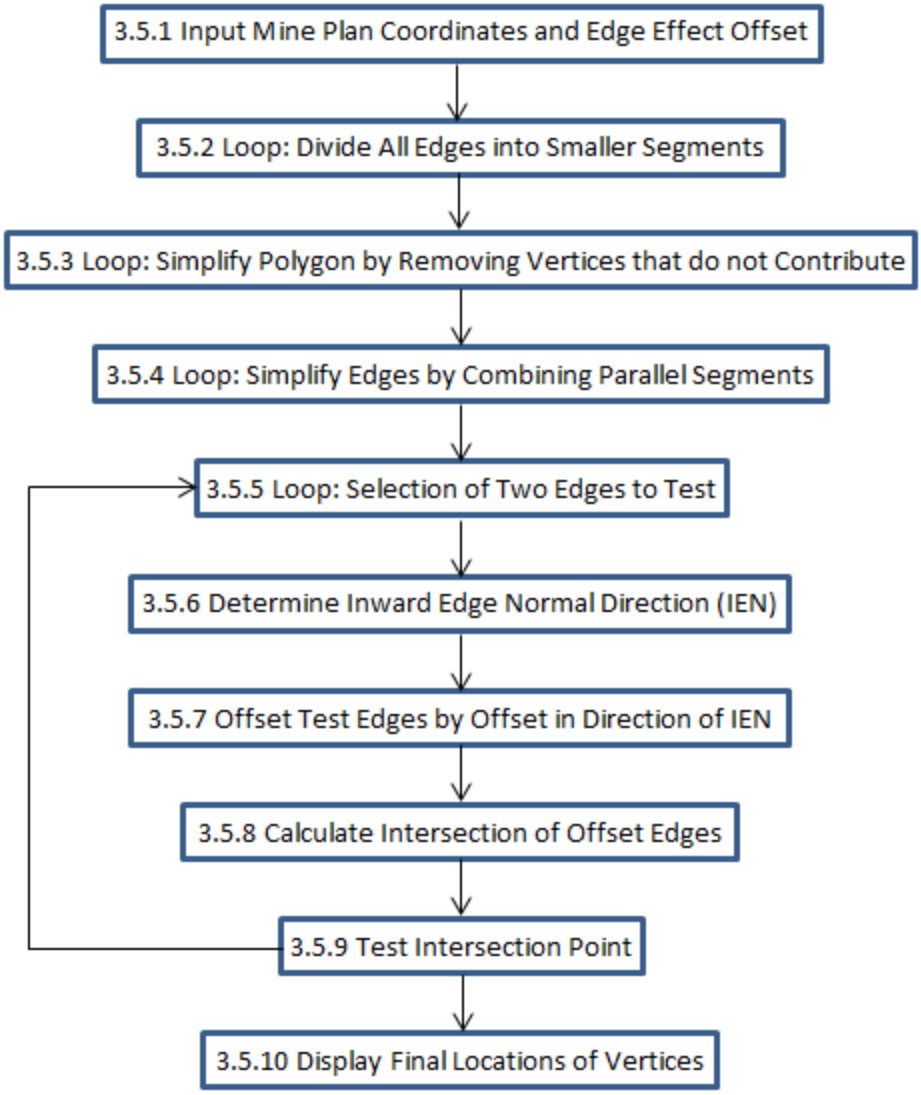

Figure 12: Flowchart of the methodology of the offset algorithm.

The first steps involve the input of coordinates for the polygonal mine plan, as well as the edge effect offset distance (d) (i.e., the distance the polygonal mine plan will be offset inward). The line segments between the coordinate points are divided into smaller segments to increase the accuracy of the next step. The mine plan is simplified through the use of an algorithm that will delete coordinates that match certain conditions. After the polygon has been simplified the smaller segments are combined into larger segments to reduce the total number of vertices that the algorithm must process.

The algorithm then uses a loop function to pick two polygon edges to perform calculations. Using the two test edges the algorithm uses a series of geometric equations to determine the direction of the inward edge normal (IEN) for each test edge, translate each test edge a distance equal to the edge effect offset in the direction of the IEN. The

algorithm then calculates the intersection point of the two offset edges. The intersection point is then tested to determine if all of the distances between that intersection point and all original edges and vertices is greater or equal to the input offset distance and to test if the point is inside of the original polygon. The final test on the intersection point determines if the point is part of a small acute angle and calculates two new coordinates to replace the intersection point when this occurs. Either the intersection point or the two replacement coordinates are saved if the tests performed on the intersection point are passed.

Figure 13 provides an example of one of the distance tests. The plotted circles at each original vertex verify that the distance between any of the offset vertices and any of the original mine plan vertices is equal or greater than the value of the edge effect offset that the user chose. After the two distance tests are performed, any vertex point determined to be too close to the original mine plan ribs is deleted. The final list of coordinates is given and the polygon of the correct offset mine plan is displayed for the user. A much more detailed explanation of the algorithm is given in Section 3.5.

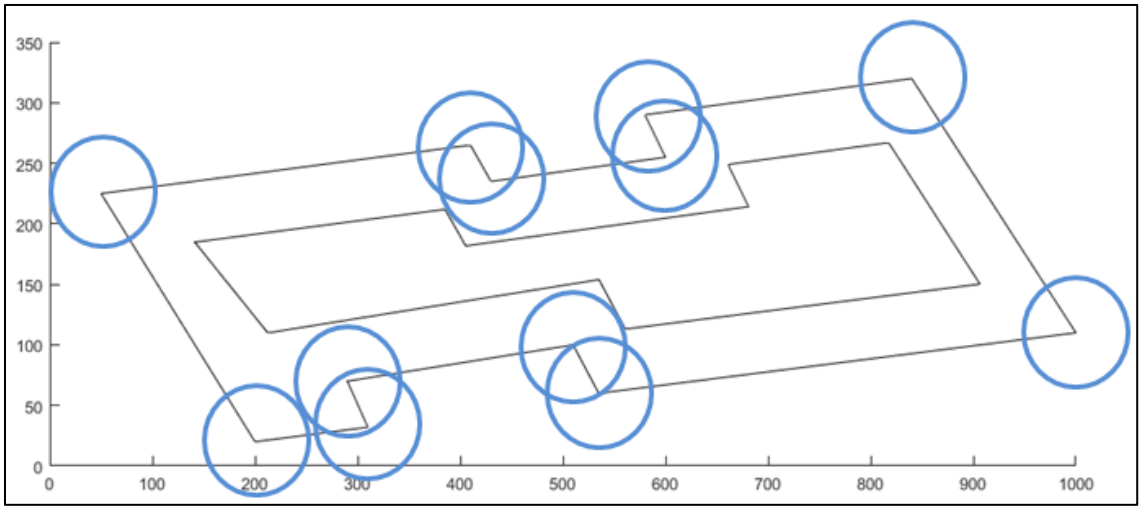

Figure 13: Circles of radius, d, show proper distance between mine plan and offset polygon.

3.5 Analysis of Individual Functions

3.5.1 Input Mine Plan Coordinates and Edge Effect Offset

The first step of the algorithm asks for the user to input the coordinates of the desired polygon. These coordinates will serve as the original polygon for the calculations and will be used to determine the location of the inward offset polygon, offset by the input edge effect offset distance. The first step also requests the user to input the desired edge effect offset distance to use in the calculation of the inward offset polygon. This value is normally pre-calculated within SDPS based on a width-to-depth ratio in association with the width of the extracted panel and the depth of cover above the panel.

3.5.2 Loop: Divide All Edges into Smaller Segments

To increase the resolution of the step involving the removal of non-contributing vertices, the line segments will be divided up into smaller segments. This step is performed by using a local offset distance value which is calculated within this step. To increase the resolution of the final offset polygon output, the user can divide the input edge effect offset distance by an integer and the resulting variable is called the local offset distance. The result of doing this causes the creation of more segments of smaller length as the integer value is increased by making the new line segments lengths equal to a fraction of the edge effect offset distance. This will increase the resolution of the algorithm's final output however it will result in a higher run time due to the increased number of vertices on which the algorithm will need to perform the calculations.

The length of each original polygon edge is compared to the local offset distance. If the length of the edge is larger than the local offset distance then the original edge's length is divided by the local offset distance to determine how many segments of length equal to the local offset distance would be necessary to replace the original segment. The resulting value will most likely not be an integer so the algorithm rounds the value up to the nearest integer value. The original edge's length is divided by this new integer value to determine the length that every new segment will be given in order to replace the original edge.

The first vertex of the original edge being tested is saved as the first vertex in the series of new segments being created. The first vertex saved for any given original edge can also be considered the last vertex of the adjacent edge because it is a shared coordinate point. Therefore, the algorithm will only create n-1 new vertices with n being the rounded up integer that was used to determine the length of each of the new segments. Using a FOR loop indexed j=1:n-1 the algorithm will create the new vertex locations at the calculated distances away from the first vertex's location along the same slope as the original edge. The equations used to perform this step are shown below.

$$\text{Local Offset Distance (LOD)} = \frac{\text{Edge Effect Offset}}{\text{Input Desired Integer}} \tag{3.1}$$

$$\text{Original Edge Length (OEL)} = \sqrt{(y_2 - y_1)^2 + ((x_2 - x_1)^2} \tag{3.2}$$

$$\text{Number Segments} = \text{ceiling}\left(\frac{\text{OEL}}{\text{LOD}}\right) \tag{3.3}$$

$$\text{New Segment Length (NSL)} = \frac{\text{OEL}}{\text{Number Segments}} \tag{3.4}$$

$$\text{Number New Vertices Created (n)} = \text{ Number Segments} - 1 \tag{3.5}$$

$$\text{Slope (m)} = \frac{y_2 - y_1}{x_2 - x_1} \tag{3.6}$$

$$x_{new}(j) = x_1 + \frac{NSL * j}{\sqrt{1 + m^2}} \tag{3.7}$$

$$y_{new}(j) = m * (x_{new}) + y_1 \tag{3.8}$$

Using a loop, this process is repeated for each of the original edges within the input polygon and the coordinate points created by this step are added to the list of coordinates that define the original polygon. This step only increases the number of coordinate points that define the original polygon. No change to the shape of the polygon is performed during this step.

3.5.3 Loop: Simplify Polygon by Removing Vertices that do not Contribute

The existence of narrow openings within a polygon results in difficulties for the offsetting process. As the offset distance is increased, the edges of the opening get closer to each other until they become coincident lines, parallel and overlapping. As a result, the final offset polygon should not include any indication of the opening having existed. The circles shown in each of the examples show the theoretical location of the vertices that define the polygon which were created by the previous step that created the smaller line segments.

Figure 14 shows an example polygon that has a narrow opening in the southwest corner. It also reveals the process of increasing the offset distance until the opening is no longer included within the final offset polygon, shown in blue.

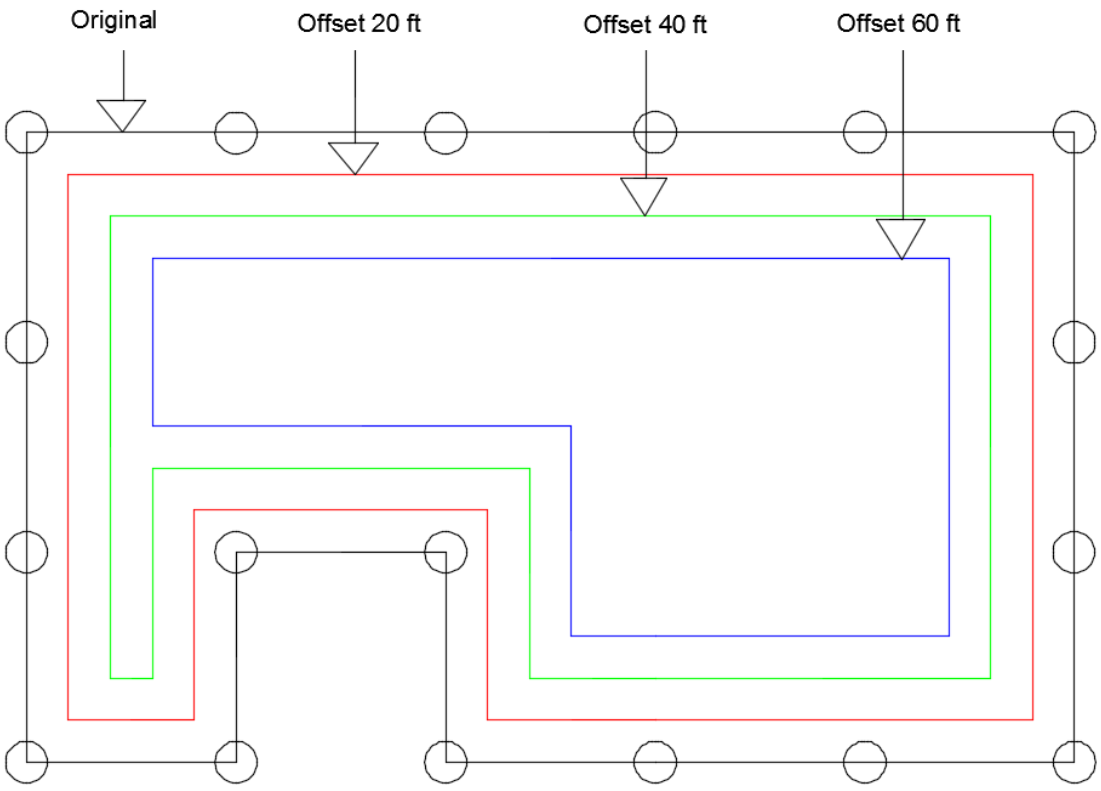

Figure 14: Polygon vertices shown with circles and multiple offsets performed.

Figure 15 contains the same original polygon however the opening in the southwest corner has been omitted from the polygon before the offsets are performed on the polygon. The dashed line represents the edge that replaces the southwest opening and the offset is performed again. The red and green offset lines are different from those in Figure 14; however the inner offset line, blue, is the exact same as the blue offset in Figure 14.

This reveals that if the chosen offset distance will cause narrow openings to be excluded from the final offset polygon then the opening can be omitted from the original polygon before performing the offset and the final result will be the same as if it had been left in. The difference between the two methods is that redefining the original polygon to exclude these openings will simplify the original polygon and reduce the likelihood of any errors occurring during the offset procedure.

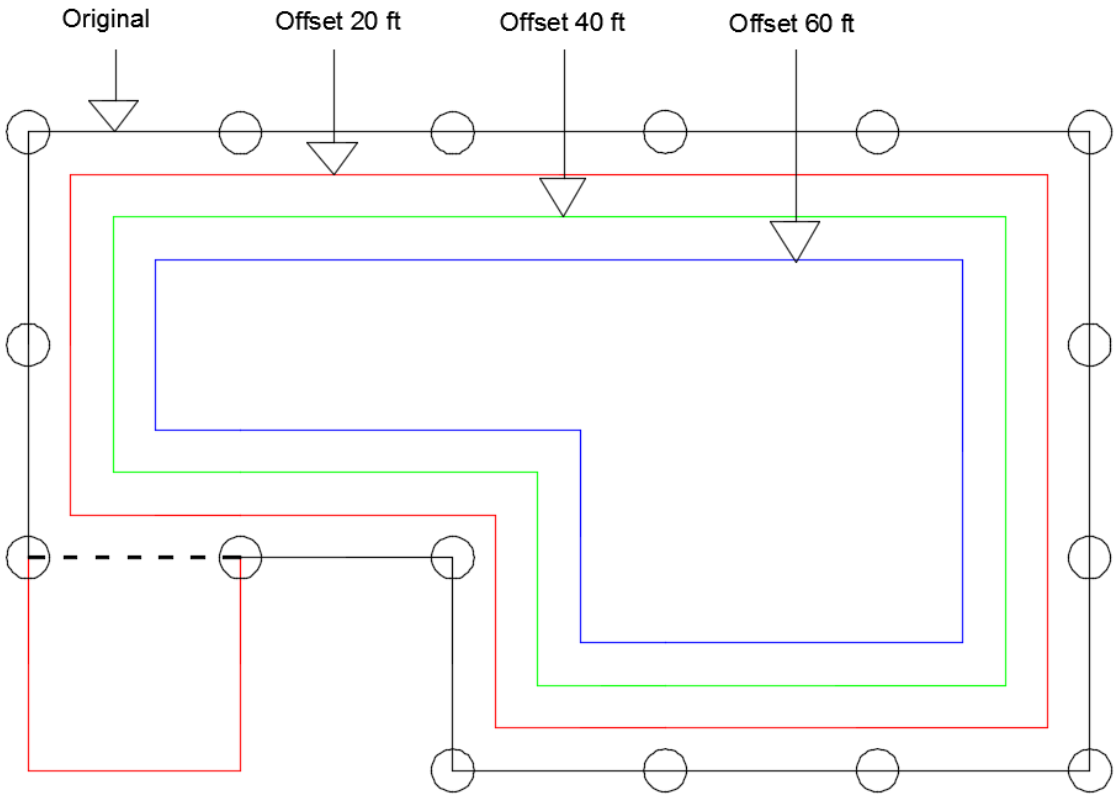

Figure 15: Polygon redefined with new vertices and multiple offsets performed.

Examining this methodology with a second example, Figure 16 shows a polygon with a narrow opening located within the southern edge. The red and green offset lines include the edges of the opening in the final offset polygon; however, the blue offset line does not because the offset distance is too large. The same methodology mentioned previously will be performed on this polygon as shown in Figure 17. The original polygon is redefined so that the edges of the southern area are excluded from the polygon and the dashed line represents the edge that replaces the southern opening. The red and green offset lines are different from those in Figure 16; however the inner offset line, blue, is the exact same as the blue offset in Figure 16.

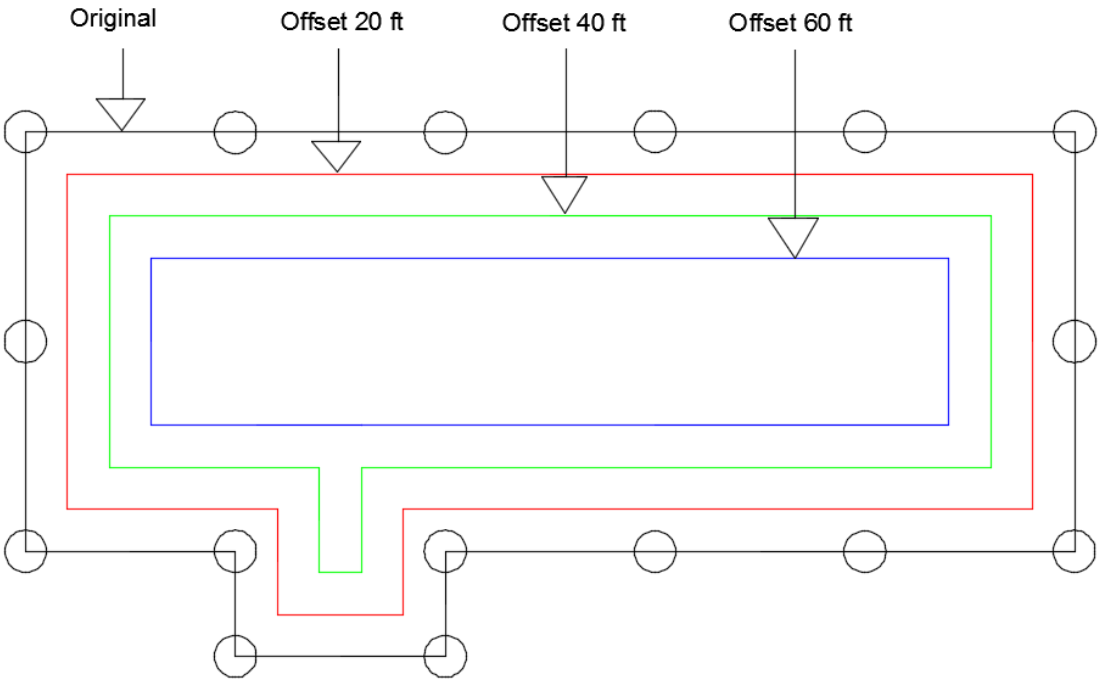

Figure 16: Polygon vertices shown with circles and multiple offsets performed.

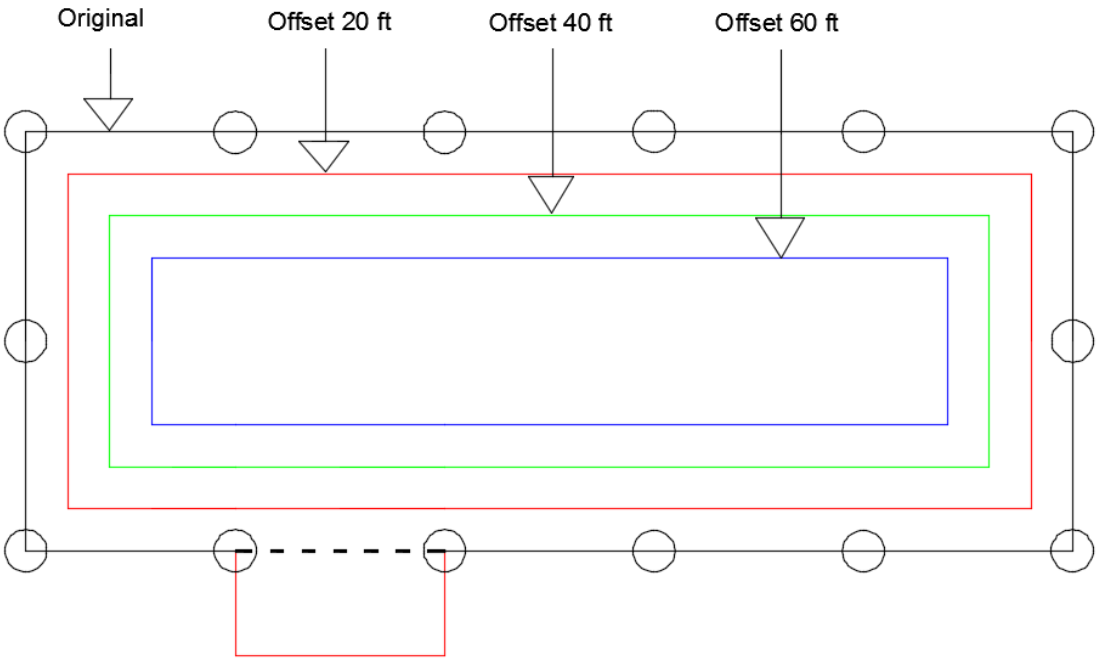

Figure 17: Polygon redefined with new vertices and multiple offsets performed.

This result agrees with the results from Figure 14 and Figure 15. If it can be determined before performing the offset routine that one or more narrow openings will be excluded from the final offset polygon, then the original polygon can be redefined to exclude these openings, resulting in a simpler polygon. The more geometric a polygon is before the offset is performed the less likely that any errors will occur in calculating the final offset polygon.

This step uses a WHILE loop to choose three vertices in series to test if the opening matches the conditions previously discussed. Vertices that are determined to match the conditions mentioned previously are marked for deletion and cannot be used as a test vertex in any of the following iterations of the loop.

The inward edge normal (IEN) is calculated for the edge defined by vertex one and two and for the edge defined by vertex two and three followed by the midpoint of each edge. The midpoint and IEN direction of each edge is used to form a ray and the intersection of the two rays is calculated.

$$u = \frac{as.y * bd.x + bd.y * bs.x - bs.y * bd.x - bd.y * as.x}{(ad.x * bd.y)} \tag{3.9}$$

$$v = \frac{(as.x + (ad.x * u) - bs.x}{bd.x} \tag{3.10}$$

Where:

as = midpoint for the edge defined by vertices one and two
ad = direction vector (IEN) for the edge defined by vertices one and two
bs = midpoint for the edge defined by vertices two and three
bd = direction vector (IEN) for the edge defined by vertices two and three

If the values of u and v are both positive, then the rays intersect inside the input polygon.

$$intersect.x = as.x + (ad.x * u) \tag{3.11}$$

$$intersect.y = as.y + (ad.y * u) \tag{3.12}$$

The distance between the intersection and each of the midpoints is calculated. If one or both of the calculated distances between the intersect coordinate and the midpoints is less than the input edge effect offset distance, the vertex chosen as the second test point is marked for removal from the coordinate matrix.

$$\text{distance} = \sqrt{(\text{intersect.x} - \text{midpoint.x})^2 + (\text{intersect.y} - \text{midpoint.y})^2} \qquad (3.13)$$

If one or both of the distances is too small the algorithm will repeat the process again by choosing the same first test vertex, skips the vertex that was marked for deletion, and then choose the next available vertices for the second and third test vertices.

If both of the distances are greater than or equal to the input edge effect offset distance then no coordinate is marked for deletion and the next set of three test vertices is chosen by using the second test vertex from the previous iteration as the first test vertex of the new iteration and selecting the next two available coordinates for the second and third test vertices.

These steps are repeated until the algorithm is unable to select three test vertices for the next iteration. This occurs because all possible sets of three test coordinates have been tested and all vertices that meet the deletion conditions have been marked for deletion. The algorithm then identifies all vertices marked for deletion and removes them all at the same time at the end of this step.

3.5.4 Loop: Simplify Edges by Combining Parallel Segments

The large number of line segments created in 3.5.2 was necessary to reduce errors in the offset calculations as non-contributing vertices were removed in 3.5.3. Dependent on the new length of the new line segments, the large number of vertices created can lead to longer calculation times due to the use of WHILE and FOR loops within the algorithm.

To reduce the number of edges and vertices that the algorithm must process, this step will determine if two segments in series are parallel to each other. When these conditions are met the algorithm will remove the vertex that is shared by the two segments from the coordinate matrix. This results in combining two segments into one. This is performed using a FOR loop to check all segments against adjacent segments.

This step serves a secondary purpose as well. When calculating the locations of the offset polygon vertices, the algorithm will examine two edges that are in series. If the two edges are in parallel with each other, the algorithm will not be able to perform the calculations for the two edges. By simplifying parallel edges that are adjacent to each other into a single segment, this step removes the possibility of errors in future calculations due to the presence of parallel segments.

3.5.5 Loop: Selection of Two Edges to Test

This step uses a WHILE loop to choose two vertices in series to perform a series of calculations in order to determine the locations of the vertices of the inward offset polygon. These calculations are described in Sections 3.5.6 - 3.5.9. This is performed by

using the first test vertex paired with the next vertex in the coordinate list to define the first edge and the second test vertex paired with the next vertex in the coordinate list to define the second edge. For example, if the first test vertex is the fourth coordinate then the first edge is defined by the fourth and fifth coordinate. If the second test vertex is the sixth coordinate then the second edge is defined by the sixth and seventh coordinate.

The test vertices are chosen in order as listed in the polygon coordinate list. If the intersection point using the edges defined by two test coordinates and their adjacent pairs is determined to be correct, it is saved as a final offset polygon vertex. This occurs when the intersection point is located inside the original polygon and is a distance equal to or greater than the input edge effect offset distance from all original edges and vertices. If the intersection point fails these conditions the intersection point is not saved and the algorithm chooses the next set of test vertices.

The selection of the two test coordinates is repeated to test if the intersection points are correct until the algorithm is unable to select two test coordinates, which occurs at the end of the coordinate list that defines the original polygon.

3.5.6 Determine Inward Edge Normal Direction (IEN)

This step calculates the direction of the Inward Edge Normal (IEN) unit vector for the two test edges chosen by the previous step. The calculation of the IEN unit vector is performed with a series of equations:

$$\mathrm{dx} = \mathrm{vertex2_x} - \mathrm{vertex1_x} \tag{3.14}$$

$$\mathrm{dy} = \mathrm{vertex2_y} - \mathrm{vertex1_y} \tag{3.15}$$

$$\text{edge length } = \sqrt{\mathrm{dx}^2 + \mathrm{dy}^2} \tag{3.16}$$

The IEN calculated for a polygon whose vertices are listed in counter-clockwise order:

$$\mathrm{IEN_x} = \frac{-\mathrm{dy}}{\text{edge length}} \tag{3.17}$$

$$\mathrm{IEN_y} = \frac{\mathrm{dx}}{\text{edge length}} \tag{3.18}$$

The IEN calculated for a polygon whose vertices are listed in clockwise order:

$$\mathrm{IEN_x} = \frac{\mathrm{dy}}{\text{edge length}} \tag{3.19}$$

$$\mathrm{IEN_y} = \frac{-\mathrm{dx}}{\text{edge length}} \tag{3.20}$$

Figure 18 shows an example of how the IEN is performed. A vector of unit length equal to one is drawn perpendicular to the line segment it is being performed on. For line segments that are vertical or horizontal, the unit vector will only have an x or y component respectively, as seen in Figure 18. For simplicity, the IEN unit vectors are drawn from the midpoint of each line segment, however it is only necessary to calculate and record the x and y components of the unit vector for proceeding steps. The edge normal arrows are considered to be positive if they point into the polygon and negative when they point out of the polygon, when the coordinates of the polygon are listed counter-clockwise wise around the polygon.

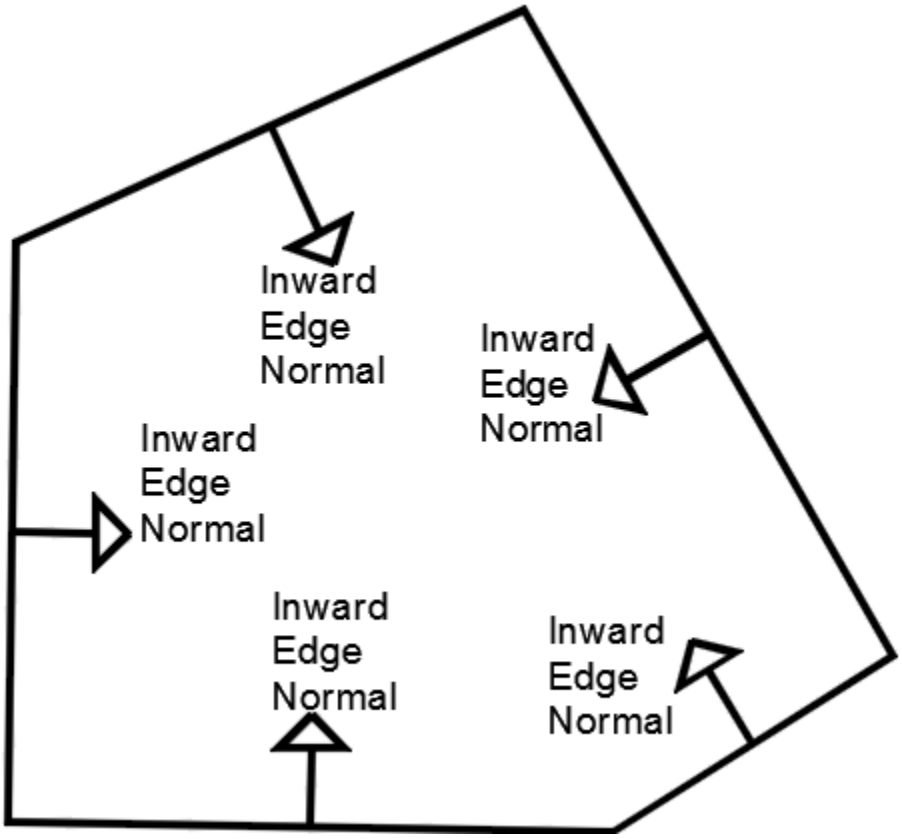

Figure 18: Inward edge normal example.

3.5.7 Offset Test Edges by Offset in Direction of IEN

This step uses the IEN unit vector values calculated for the two test edges in the previous step and the edge effect offset distance input by the user in the first step. The algorithm multiplies the edge effect offset distance with the IEN_x and IEN_y values for each test edge in order to calculate the distance that each vertex will be offset in the direction of its respective IEN. It is important to recognize that the methodology of this step is to offset line segments as opposed to offsetting the vertices. As a result, if a vertex is shared by the two chosen test edges then the shared vertex will effectively be translated two separate times to account for the specific magnitude and direction attributed to each of those two test edges. An example calculation for the first test edge is shown below:

$$\text{edge1\_x1} = \text{vertex1}_x + (\text{offset\_distance} * \text{IEN\_1\_x}) \tag{3.21}$$

$$\text{edge1\_y1} = \text{vertex1}_y + (\text{offset\_distance} * \text{IEN\_1\_y}) \tag{3.22}$$

$$\text{edge1\_x2} = \text{vertex2}_x + (\text{offset\_distance} * \text{IEN\_1\_x}) \tag{3.23}$$

$$\text{edge1\_y2} = \text{vertex2}_y + (\text{offset\_distance} * \text{IEN\_1\_y}) \tag{3.24}$$

The example above is repeated for the second test edge, resulting in a total of four new vertex locations. These four coordinates are used by the next step to determine the location of the intersection of the offset edges.

3.5.8 Calculate Intersection of Offset Edges

The algorithm then uses the four new coordinates created as well as geometric equations in order to determine the location that the offset edges will intersect. The calculated intersection will be examined afterwards to determine if it is correct with regards to its location compared to the "condensed" polygon. The equations used in this process are as follows:

$$\text{dd} = [(\text{y4} - \text{y3}) * (\text{x2} - \text{x1})] - [(\text{x4} - \text{x3}) * (\text{y2} - \text{y1})] \tag{3.25}$$

$$\text{n}_a = [(\text{x4} - \text{x3}) * (\text{y1} - \text{y3})] - [(\text{y4} - \text{y3}) * (\text{x1} - \text{x3})] \tag{3.26}$$

$$\text{u}_a = \frac{\text{n}_a}{\text{dd}} \tag{3.27}$$

$$\text{intersect}_x = x1 + [u_a * (x2 - x1)] \quad (3.28)$$

$$\text{intersect}_y = y1 + [u_a * (y2 - y1)] \quad (3.29)$$

If the value of "dd" is zero, the two offset edges are parallel and will not intersect. The step that simplified parallel edges in series into a single line segment has already removed the possibility of this happening. The final two equations utilize u_a and the vertices of the first line segment used within the calculation to determine the coordinates of the raw offset vertex created from these two line segments.

It is important to note that the equations used above treat line segments as infinite lines. This results in the calculation of an intersection point between two line segments regardless of whether they intersect or must be extended beyond their defined limits in order to find the intersection.

3.5.9 Test Intersection Point

The final step of the WHILE loop will test the validity of the offset intersection point using a series of tests: check the distance between the intersection point against all original polygon vertices, check the distance between the intersection point against all original polygon edges and to determine if the intersection is inside the original polygon. If the intersection point passes all of the tests then it is considered correct. A final calculation will determine if the two test edges meet at a sharp angle, which creates a large triangular offset as previously discussed in the AutoCAD examples: Figure 7 and Figure 8.

If the distances between the intersection point and one or more of the original polygon vertices are less than the input edge effect offset distance, then it is incorrect and the algorithm chooses the next set of test edges. If all of the distances between the intersection point all the original polygon vertices is equal to or greater than the input edge effect offset, then the intersection point will be tested further.

Two tests are necessary to test the distance between the intersection point and all of the original polygon edges. The first distance test calculates the shortest distance between the intersection point and each of the original mine plan line segments using a FOR loop. If the shortest distance between the intersection point and any line segment on the original polygon is less than the user input edge effect offset distance, then the intersection point is marked to be tested by the second distance test.

The equation used is as follows:

$$\text{Distance} = \frac{(y2 - y1)\text{intersect\_x} - (x2 - x1)\text{intersect\_y} + x2y1 - y2x1}{\sqrt{(y2 - y1)^2 + (x2 - x1)^2}} \quad (3.30)$$

It is important to recognize that this equation treats the designated original edge as an infinite line, which results in cases where an intersection point will be considered too close to an original edge even though the intersection point is located outside the defined boundaries of the original edge.

A second distance test was created in order to more accurately calculate the distance between the calculated intersection point and the original edges and while saving intersection point that were going to be deleted as false positives of being too close to an original edge. A modified intersection equation was used that utilizes the negative inverse of the slope of the original edge that an intersection point is being tested against. The equations used in this test are as follows:

$$\text{negative inverse slope} = -\frac{x2 - x1}{y2 - y1} \quad (3.31)$$

$$y = m * x + b \quad (3.32)$$

The negative inverse of the slope is substituted in for the slope and equation 3.24 becomes:

$$y = \text{negative inverse slope} * x + b \quad (3.33)$$

Rearranging equation 3.25 to solve for parameter "b":

$$b = y - \text{negative inverse slope} * x \quad (3.34)$$

The negative inverse of the slope of the line segment is used in conjunction with the intersection point's coordinates in order to create a line which is perpendicular to the original edge and travels through the intersection point. This line represents the shortest distance between the intersection point and the original edge. The line segment created has two vertices: one with the coordinates of the intersection point being tested and the other given as (0,b). The parameter "b" is the y-intercept of the line that is perpendicular to the original edge and travels through the intersection point being tested.

The four vertices are used to calculate the location of the intersection between the line perpendicular to the original edge and the original edge itself. The location of the intersection is defined by the coordinates "x_new" and "y_new". If this new intersection point is located outside of the limits of the original edge, then the intersection point

calculated was given a false positive by the first distance test and is saved from deletion. If the resulting intersection point is located inside the limits of the line segment then the intersection point is correctly deleted and the algorithm chooses the next set of test edges.

Using the negative inverse of the slope of the edge the intersection point is being checked against results in an undefined value when taking the inverse slope of a horizontal line. In these cases the algorithm recognizes that a horizontal edge is being examined and tests if the intersection point's x-coordinate is outside of the horizontal line's boundary limits. If it is outside the bounds then the point is saved from being deleted because it was a false positive in the first distance test. If it is inside the bounds, it is deleted because it is too close to an edge of the condensed polygon.

The next test checks to see if the raw intersection point is inside or outside of the condensed polygon. If it is outside of the condensed polygon then it is incorrect and the algorithm chooses the next set of test edges. If the intersection point is determined to be inside the condensed polygon then it is correct and saved as a final offset vertex. Figure 19 shows multiple examples of this test.

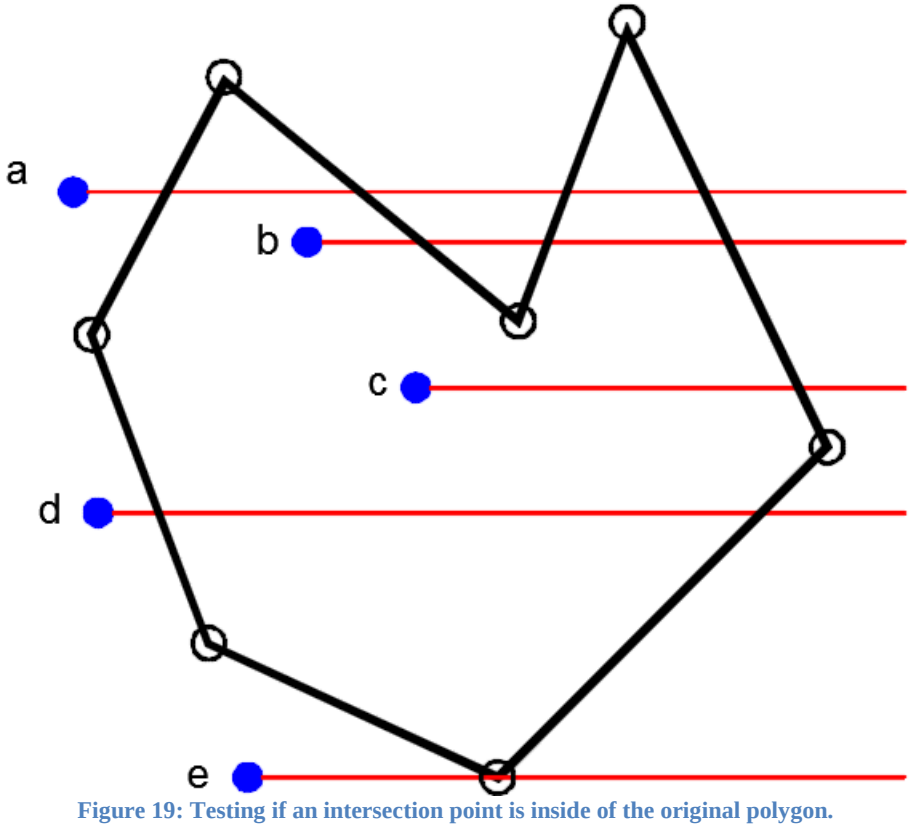

Figure 19: Testing if an intersection point is inside of the original polygon.

This test is performed by counting the number of intersection points between a horizontal ray defined by the offset intersection point and extending to positive infinity and all of

the condensed polygon edges. The intersection equations used within the algorithm calculate the intersection of two line segments as if they are infinite lines. Therefore it was only necessary to define the horizontal line using the raw intersection coordinate and a second coordinate located one unit to the right of the intersection point. The methodology of this step requires that an intersection between the horizontal line and an edge only be counted if it occurs to the right of the offset intersection point being tested for correctness. In addition, the intersection of the ray and a vertex on the original polygon is not counted as shown in Figure 19, example "e".

The algorithm determines the total count of intersections that match the conditions necessary for an intersection to be counted. If the total number of intersections is an odd integer, the offset intersection coordinate is located inside of the condensed polygon. If the total number of intersections is zero or an even number, the offset intersection point is located outside of the condensed polygon and fails the test. The algorithm then marks the second vertex used for the iteration as exempt from future iterations and selects the next available two edges to be offset.

The final test determines if the two test edges form a sharp angle. As discussed in Figure 7 and Figure 8, a sharp angle can result in splitting the final offset polygon into two polygons when the input edge effect offset distance is large enough. This test is necessary to identify if the two test edges will make a sharp angle and will potentially cause the final polygon to be split in two. This final test will only be performed if the calculated intersection point passed all of the previous tests. The following explanation of this test is visualized in Figure 20.

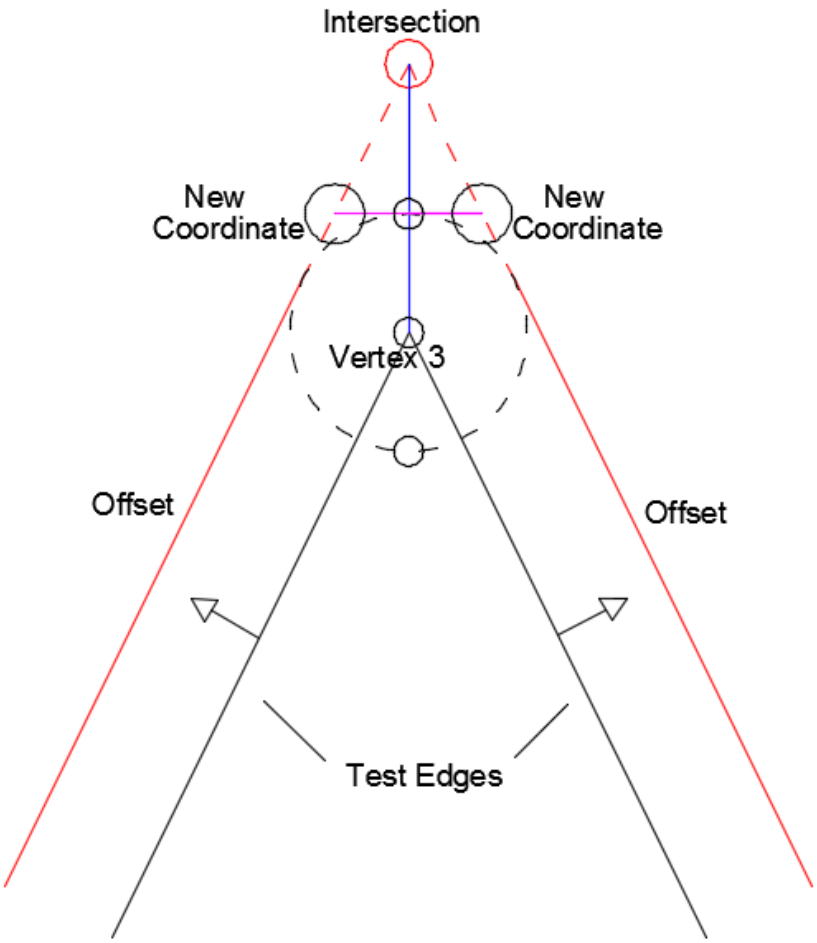

Figure 20: Example calculation of two new coordinates for an arrowhead.

The first part of this test calculates the midpoint for the two test edges before they are offset. Using the midpoint and the previously calculated IENs, the values of u and v are calculated for the two edges to determine if the rays formed from the midpoints and IENs would intersect, as seen previously in section 3.5.3. The purpose of this step is to find rays that will not intersect, when the value of u and/or v is negative which signifies the presence of an arrowhead. If the values of u and v are both positive then the rays would intersect and the intersection point previously calculated is correct and saved as a final offset coordinate.

If the rays would not intersect, the next step is to calculate the distance between the intersection point and the first vertex of the second test edge (vertex 3). If the distance between them is greater than the input offset distance multiplied by a constant greater than one, then the test edges form an arrowhead whose angle is sharp enough to potentially cause the final offset polygon to be split into two polygons. If the distance between the two points is less, then the intersection point is correct because the arrowhead does not have a sharp enough angle to cause potential problems.

The next step determines the equation of the line (vertical line shown in blue) that connects the intersection point to the vertex 3 because vertex 3 is located on the point of the arrowhead. Then the intersections between this new line and a circle whose center is vertex 3 and with radius equal to the offset distance are calculated which is shown as the black circle with the dashed line. The algorithm determines the intersection that occurs on the line previously defined which is marked by a circle between vertex 3 and the intersection point.

The algorithm then calculates the negative inverse slope of the blue line and uses the intersection calculated in the previous step to create the line equation for the line shown in pink. The intersections between the pink line and each of the offset edges are calculated in order to create the two coordinates marked as “new coordinate”.

The result of this test is to determine if an arrowhead between two test edges has a sharp enough angle to potentially split the final offset polygon into two polygons. If the two test edges meet the criteria of the previously mentioned steps then the intersection point is considered incorrect and is replaced by the two “new coordinates” in the final offset polygon matrix. If the two test edges fail any of the steps within the final test then the intersection point is considered correct and added to the final offset polygon matrix.

3.5.10 Display Final Locations of Vertices

Once the WHILE loop has completed testing the intersections of offset edges and saved all intersections that passed the previously described tests, a final matrix of the coordinates of the offset polygon is produced and the polygon is graphed for the user to examine.

4 Case Studies

4.1 General Discussion

Case studies of the algorithm are required in order to test the accuracy of the updated algorithm to insure that it performs the inward edge offset of real mine panels properly and with a high level of accuracy. The polygons used for these case studies were taken directly from the mine plan shown in Figure 21 and the coordinates of these polygons were directly input into the updated algorithm. Figure 21 Shows the locations of each of the five case studies which are depicted in green. The polygons were chosen for their unique shapes which will assess the ability of the algorithm to correctly offset polygons that have tight rooms as well as pillars that protrude into the panel.

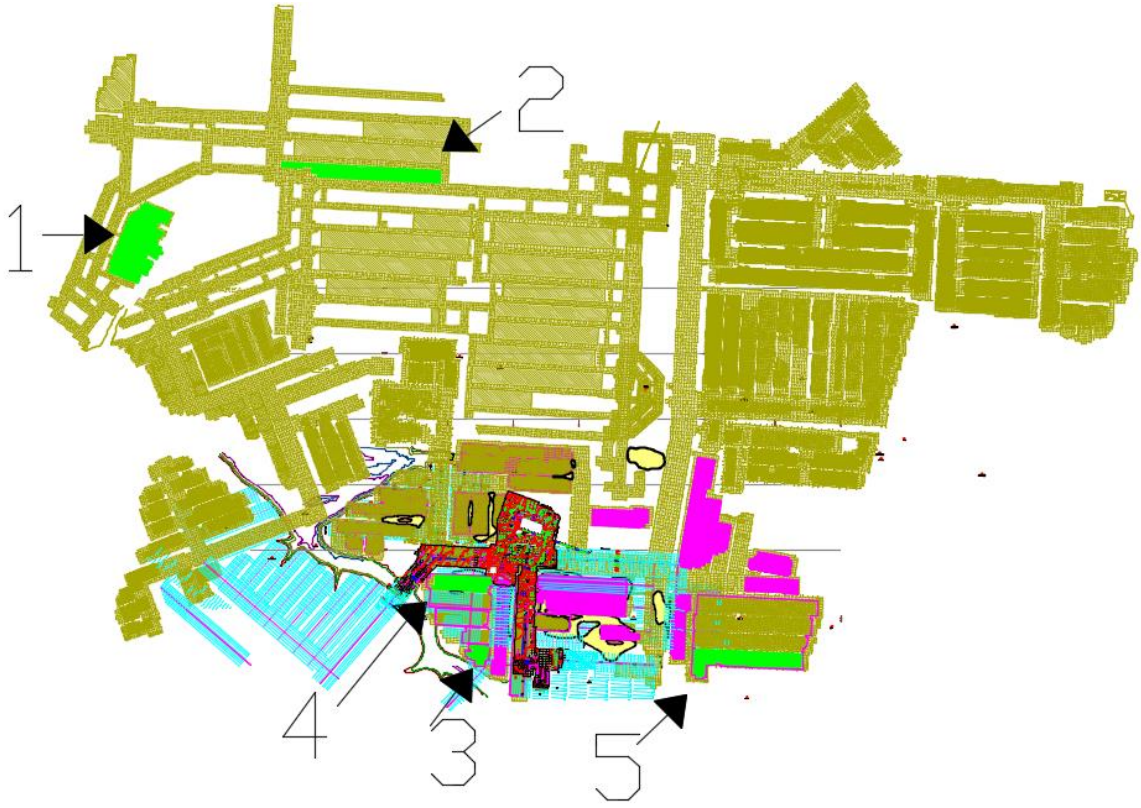

Figure 21: Mine plan used for case studies

Once chosen for a case study, a methodology was used in order to create the polygon that represents each mine panel. The calculations performed when estimating the subsidence and horizontal strain of a mine panel require the locations of the panel ribs to be known. Therefore, the polylines were drawn to include the last open entry within the panel under the assumption that any pillars located outside of the polyline would act as the ribs for the panel. Some of the case study mine panels include additional cuts in the pillars in order to increase extraction rates. In these cases the polyline followed the cut within the pillar

because what remains of the pillar will continue to act as the rib in that specific location. This methodology was performed for each of the case studies in order to maintain consistency when calculating the estimated subsidence and horizontal strain for the panel when considering an edge effect offset or no edge effect offset.

AutoCAD has an offsetting command which will offset a selected polyline by a user-defined distance in the direction chosen by the user. In the case of a polygon, created from a closed polyline, the offset can be directed to be an inward or an outward offset. In order to test the accuracy of the updated algorithm, the results of the new methodology will be compared against the offset command that AutoCAD already has established and has proven to be accurate.

4.2 Case Study 1

The polygon for the first case study was chosen due to the large occurrence of pillars that protrude into the panel, as shown in Figure 22. The large number of pillars will test the updated algorithm on its ability to handle the negative spaces that are created when offset line segments overlap each other.

The edge effect offset distance chosen for this case study was 80 feet. Figure 22 shows the original mine panel (black polygon) from Case Study 1 and the results of offsetting the mine panel by 80 feet as performed by AutoCAD's offsetting function (red polygon) and the new algorithm (blue polygon).

The result from AutoCAD's offsetting function reveals that the methodology attempts to follow the contours of the original polygon as closely as possible. In contrast, the offset polygon created by the algorithm, shown in blue, follows the contours of the mine panel less strictly and appears to flow around the contours of the mine panel. This is considered an organic polygon as opposed to a geometric polygon. This is the result of introducing a chamfer step within the algorithm which serves to remove the tips from corners that point inside or outside of the offset polygon. The purpose of the chamfering is to slightly resemble the effect of using arcs to connect edges which would be the more accurate when performing an offset routine. As explained previously in the thesis, the use of arcs was avoided in the algorithm due to the increased level of difficulty required in using arcs.

One noticeable difference between the offset polygons from AutoCAD and the new algorithm is the presence of large triangles within the AutoCAD offset polygon that protrude far into the offset polygon. The presence of these large triangles is the same as if triangular coal pillars were left in these locations and the calculation of subsidence would treat these triangles as if there were pillars inside of the triangles. In contrast, the offset polygon produced by the new algorithm, shown in blue, turns around the sharp corners on the original mine panel that are the source of the triangles in the AutoCAD offset polygon. By doing so, the algorithm creates an offset polygon that is much more accurate

in its determination of the offset polygon than the AutoCAD offsetting function produced.

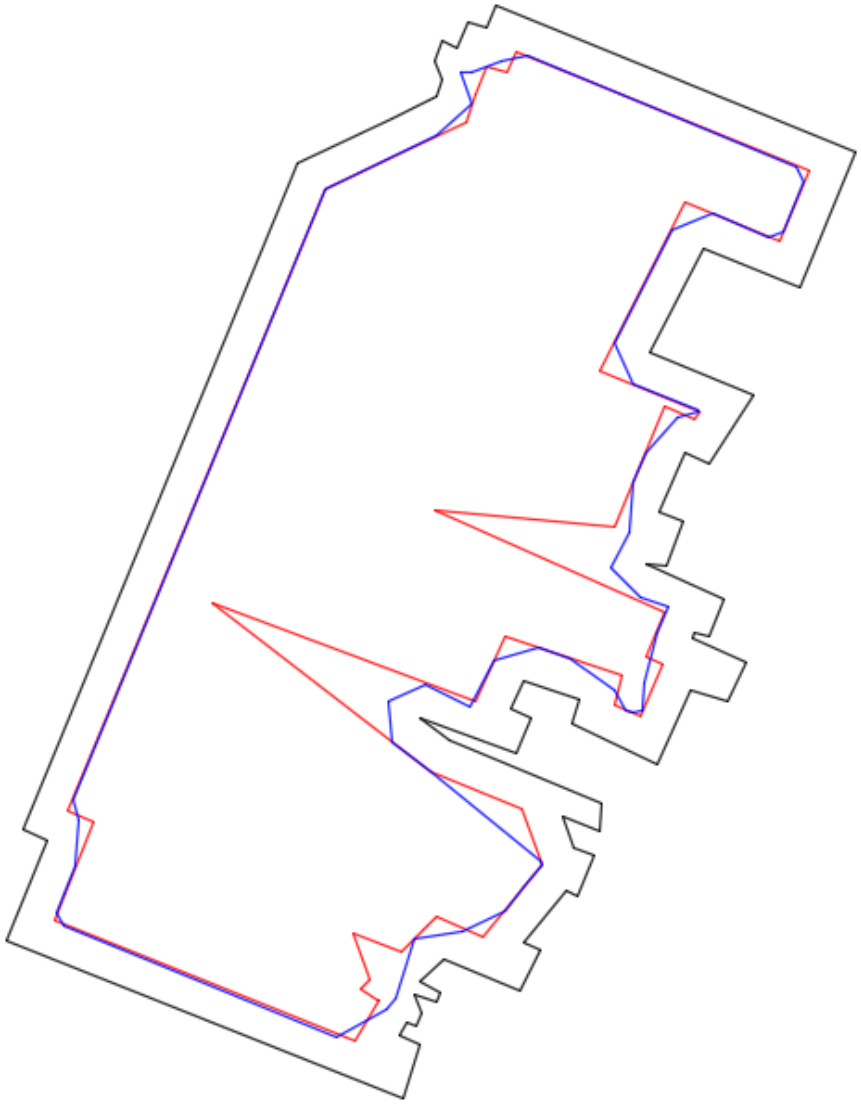

Figure 22: Case Study 1, original mine panel with offset polygons to show an offset of 80 feet from AutoCAD (red) and the new algorithm (blue).

SDPS was used for Case Study 1 to create graphs that compare the estimated subsidence and horizontal strain for the mine panel before and after the edge effect offset distance is considered. These calculations were performed using the Influence Function Method package within SDPS along with the parameters listed in Table 1.

Table 1: Influence function parameters and values used for Case Study 1.

Polygon average elevation (ft)	0
Width (ft)	2350
Prediction line average elevation (ft)	500
Average extraction thickness (ft)	6
Strain coefficient	0.35
Percent hardrock (%)	50

The panel polygon coordinates were imported directly into SDPS using the parameters previously mentioned. The depth for Case Study 1 was chosen to be 500 feet and the width was measured within AutoCAD as the largest length value of original mine panel.

To calculate the amount of subsidence and horizontal strain above the original mine panel and the two offset polygons, a prediction line was chosen as shown in Figure 23. The coordinates of the prediction line as they are input into SDPS are shown in Table 2.

Table 2: Prediction point values input into SDPS.

Minimum Easting (ft)	1,790,300
Maximum Easting (ft)	1,792,800
Cell Size in x-direction (ft)	50
Minimum Northing (ft)	417,150
Maximum Northing (ft)	417,150
Cell Size in y-direction (ft)	10
Average Point Elevation (ft)	500
Total Points	51

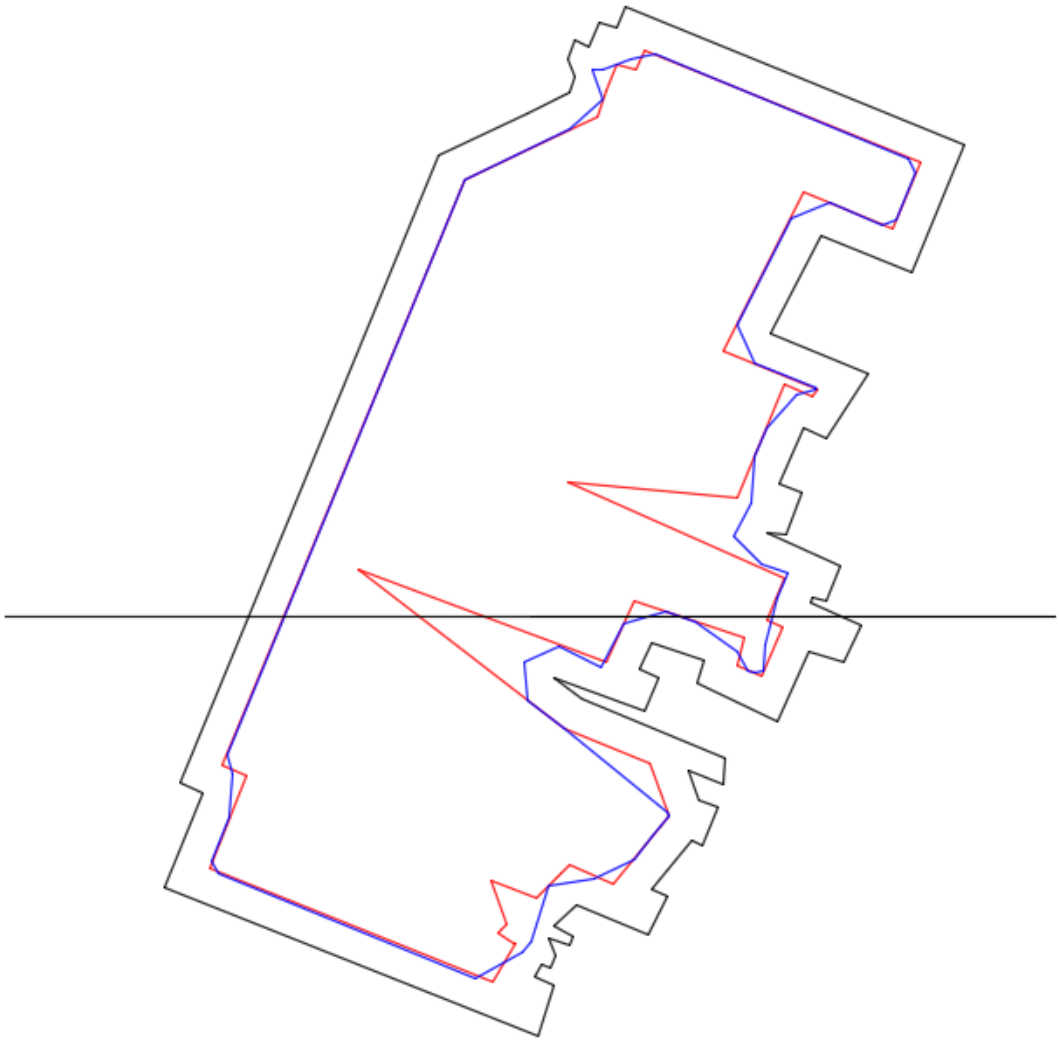

Figure 23: Case Study 1 original mine panel with offset polygons and cross-section line.

To demonstrate the impact that the edge effect offset has on subsidence and horizontal strain along the mine panel for this case study, subsidence and horizontal strain were calculated twice, once for the original mine panel with the AutoCAD offset polygon and again for the original mine panel with the new algorithm offset polygon. This allows the comparison of subsidence before and after the edge effect offset is considered and can

also be used to compare the subsidence and horizontal strain results between the AutoCAD and algorithm offsetting functions.

Figure 24 shows a graph of the subsidence results, as calculated by SDPS, for the original mine panel, the AutoCAD offset polygon, and the algorithm offset polygon.

It can be easily seen that there is a significant difference between the subsidence calculated for the original panel and the subsidence for each of the offset polygons. The difference in the amount of subsidence is a direct result of the edge effect offset distance. This parameter reveals the location of the inflection point within the panel and the result of having a non-zero edge effect offset is the reduction in the amount of effective roof that will converge to the panel floor. As a result, the lower and upper bounds of the subsidence curve for the original mine panel are pinched towards the center resulting in thinner subsidence troughs for the AutoCAD and algorithm offset polygons, as seen in Figure 24. For the algorithm offset polygon, the thinning of the subsidence trough results in a smaller area of the cross-section that reaches the same maximum value of subsidence that the original mine panel polygon has.

The lower and upper bounds of the AutoCAD and algorithm offsetting methods are identical. The noticeable difference occurs near the center of the AutoCAD curve where the subsidence is significantly less than that of the algorithm's curve. This is the result of the large triangle from Figure 22 that was discussed previously. SDPS calculated the expected subsidence for the AutoCAD offset polygon as if a triangular pillar of coal was located inside of the large triangle. As a result, the subsidence curve reflects the presence of a coal pillar which would combat the convergence of the roof during its collapse. In contrast, the subsidence curve for the algorithm offset polygon does not have the presence of a false pillar protruding into the panel, and the subsidence curve reaches nearly the same maximum subsidence value as the original panel. This is because the algorithm did not create the large triangle during the calculation.

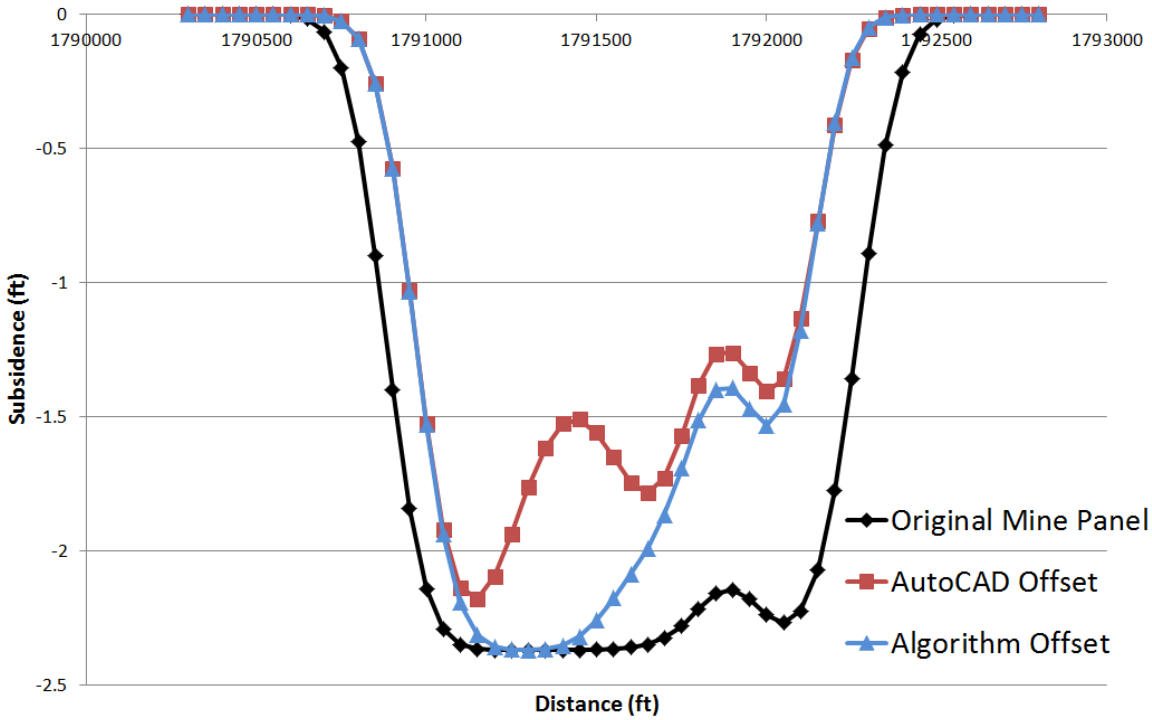

Figure 24: Subsidence for the original mine panel, AutoCAD offset and algorithm offset polygons.

Figure 25 shows a graph of the horizontal strain results, as calculated by SDPS, for the original mine panel, the AutoCAD offset polygon, and the algorithm offset polygon.

It can be easily seen that there is a noticeable difference between the horizontal strain calculated for the original panel and the horizontal strain for each of the offset polygons. The difference in the amount of subsidence is a direct result of the edge effect offset distance. The explanation for this change in horizontal strain is the same as with the changes found in the subsidence values. The edge effect offset distance results in less effective roof to converge and the result is a change in the distribution of stresses and strains.

As a result, the lower and upper bounds of the horizontal strain curve for the original mine panel are pinched towards the center resulting in a thinner horizontal strain curve for the AutoCAD and algorithm offset polygons, as seen in Figure 25. However, the thinning of the subsidence troughs also results in a different curve shape for the offset polygons, thus resulting in similar strain values in some locations and much different strain values in other locations.

The lower and upper bounds of the AutoCAD and algorithm methods are nearly identical. The noticeable difference occurs near the center of the AutoCAD curve where the horizontal strain is significantly higher than that of the algorithm's curve. This is the result of the large triangle from Figure 22 that was discussed previously. SDPS calculated the expected horizontal strain for the AutoCAD offset polygon as if a triangular pillar of

coal was located inside of the large triangle. In contrast, the horizontal strain curve for the algorithm offset polygon does not have the presence of a false pillar protruding into the panel, and the subsidence curve nearly reaches the same horizontal strain value as the original panel. This is because the algorithm did not create the large triangle during the calculation of the offset polygon.

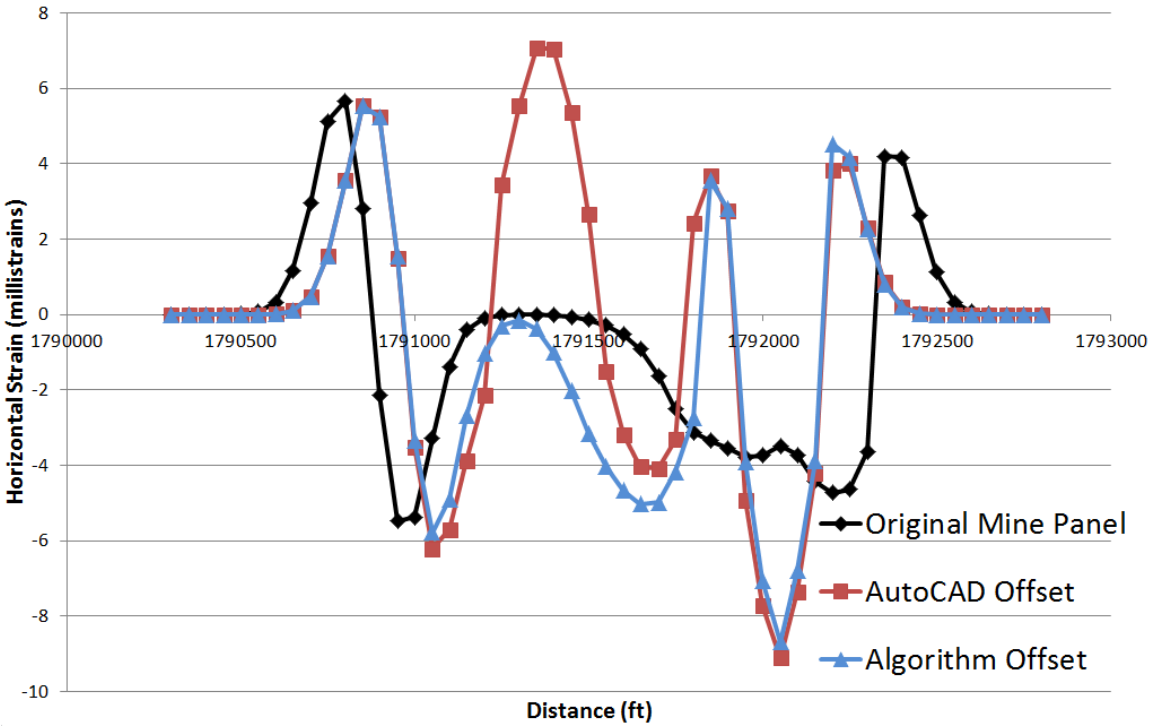

Figure 25: Horizontal strain for the original mine panel, AutoCAD offset and algorithm offset polygons.

4.3 Case Study 2

The polygon for the second case study was chosen due to its overall simplicity but still contains narrow openings and the presence of pillars that protrude into the mine panel, as shown in Figure 26. The narrow openings and pillars will test the update algorithm's ability to handle these situations accurately.

The edge effect offset distance chosen for this case study was 25 feet. Figure 26 shows the original mine panel (black polygon) from Case Study 2 and the results of offsetting the mine panel by 25 feet as performed by AutoCAD's offsetting function (red polygon) and the new algorithm (blue polygon).

The result from AutoCAD's offsetting function reveals that the methodology attempts to follow the contours of the original polygon as closely as possible. The offset polygon created by the algorithm also follows the contours of the original mine panel closely, however the chamfer function within the algorithm can be seen to remove the tips of

corners from the offset polygon. The purpose of chamfering is to slightly resemble the effect of using arcs to connect edges. As explained previously in the thesis, the use of arcs was avoided due to the level of difficulty needed to create them.

Aside from the difference in the offset polygons as a result of the chamfer function, the results from AutoCAD and the new algorithm are nearly the same. This testifies to the ability of the algorithm to accurately produce an offset polygon for near geometric mine panels.

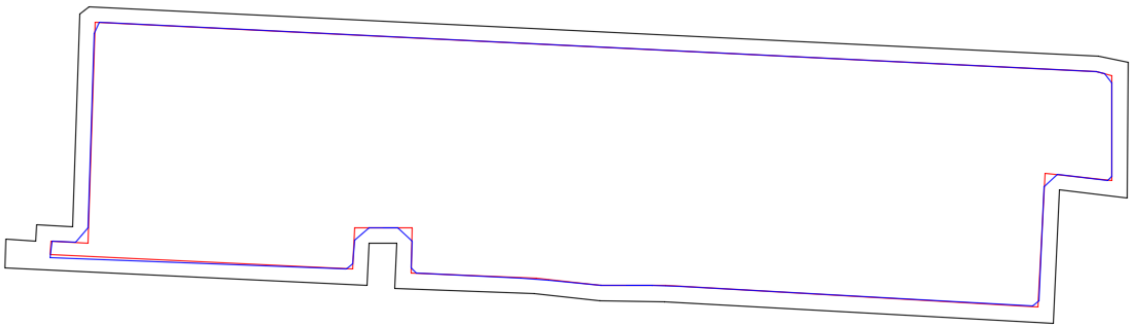

Figure 26: Case Study 2, original mine panel with offset polygons to show an offset of 25 feet from AutoCAD (red) and the new algorithm (blue).

4.4 Case Study 5

The polygon for the third case study was chosen due to the occurrence of pillars that protrude into the panel, as shown in Figure 27.

The edge effect offset distance chosen for this case study was 40 feet. Figure 27 and Figure 28 show the original mine panel (black polygon) from Case Study 3 and the results of offsetting the mine panel by 40 feet as performed by AutoCAD's offsetting function (red polygon) and the new algorithm (blue polygon). The length of the mine panel in Case Study 3 was large enough that it was necessary to split the panel in two to visualize the differences in the offsetting methods. Figure 27 is the western half of Case Study 3 and Figure 28 is the eastern half of Case Study 3.

The result from AutoCAD's offsetting function reveals that the methodology attempts to follow the contours of the original polygon as closely as possible. The offset polygon created by the algorithm also follows the contours of the original mine panel closely, however the chamfer function within the algorithm can be seen to remove the tips of corners from the offset polygon.

Aside from the difference in the offset polygons as a result of the chamfer function, the results from AutoCAD and the new algorithm are nearly the same. This testifies to the ability of the algorithm to accurately produce an offset polygon for near geometric mine panels.

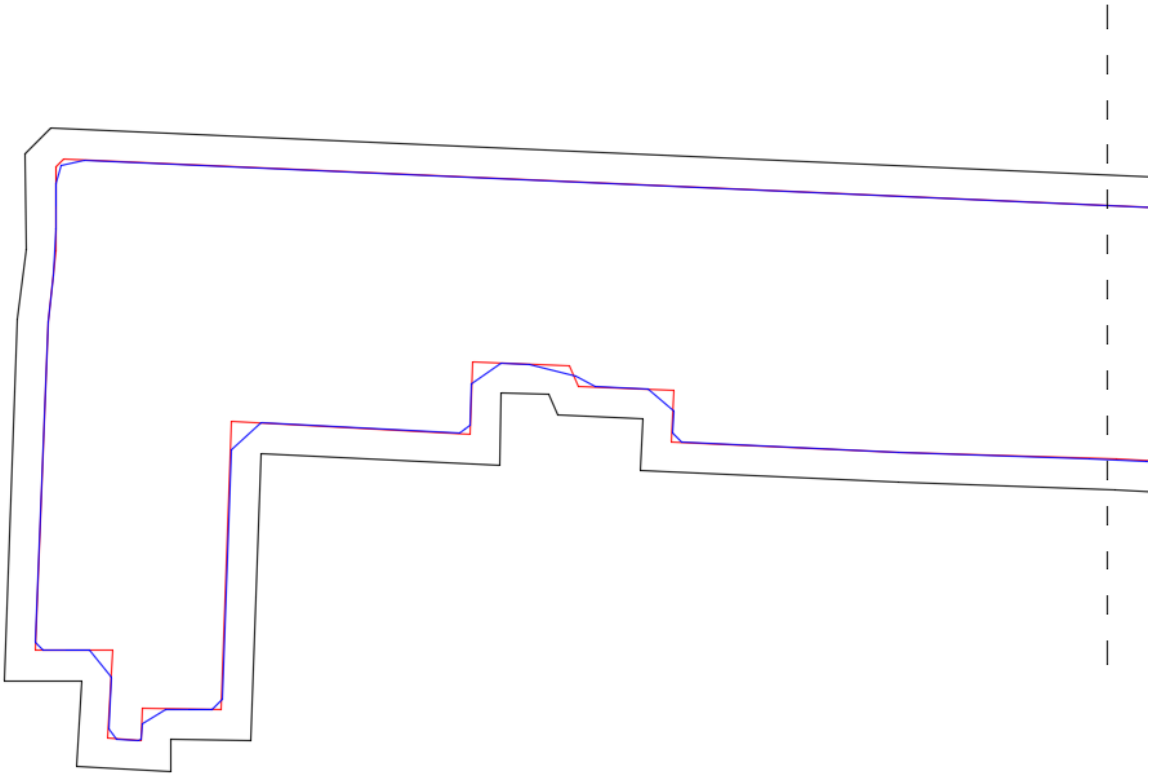

Figure 27: Case Study 3 - West, original mine panel with offset polygons to show an offset of 40 feet from AutoCAD (red) and the new algorithm (blue).

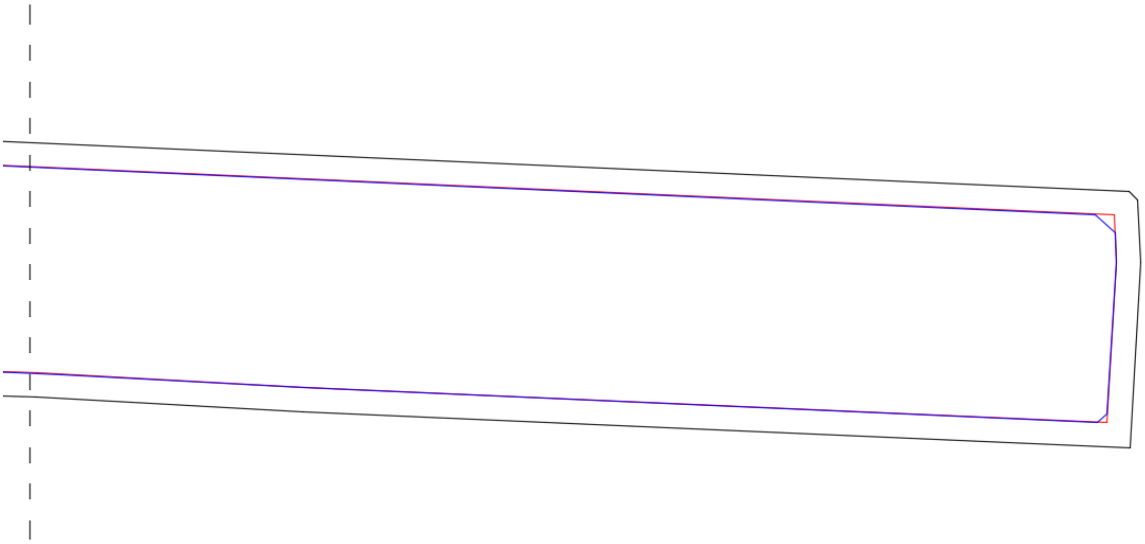

Figure 28: Case Study 3 - East, original mine panel with offset polygons to show an offset of 40 feet from AutoCAD (red) and the new algorithm (blue).

5 Conclusions and Recommendations

The case studies and the subsidence and horizontal strain charts reveal that the methodology used within the algorithm is capable of accurately offsetting a polygon inward when using AutoCAD as the standard. The differences between the methodologies area used by AutoCAD and the new algorithm and the resulting difference in the area of the inward offset polygons are minimal. However, the case studies and the graphs have revealed that the new algorithm is capable of calculating the inward offset of a polygon more accurately for the purposes of calculating surface subsidence than the default settings of the AutoCAD offsetting function.

As a part of the design of the algorithm, the level of resolution that the user desires can be adjusted during the process of splitting long polygon segments into smaller ones. By default, the algorithm will split the line segment into as many smaller segments of length equal to the input edge effect offset as is possible for the given length. Alternatively, the user can further decrease the length that the new segments are given by dividing the edge effect offset distance by an integer. During the creation and testing of the algorithm, the edge effect offset distance was divided by integers such as three, five, seven and ten to test the response of the algorithm and its accuracy to different levels of resolution.

The resulting tests have shown that dividing the edge effect offset by a larger integer will decrease the length of each of the smaller segments created and as a result will largely increase the number of vertices that define the polygon. The large number of created vertices can result in a longer run time as the algorithm must process all of the new vertices during the step that removes vertices that don't contribute and the step that condenses the list of vertices that define the polygon. The large number of vertices has no effect on the run time of steps that proceed after the condensing step.

The effect of decreasing the new length of the line segments increases the resolution of the final offset polygon. The result is a softening of sharp corners and the reduction in the amount of the original polygon that is removed during the process of removing vertices that do not contribute. Using the input edge effect offset distance as the length for the new segments results in a final offset polygon that will appear more angular with sharper corners than if smaller lengths were used. The accuracy of the final offset polygon is still maintained when using larger segment lengths while shortening the necessary runtime. It is up to the user to determine the balance between resolution and runtime desired for the polygon being tested.

Future work will include rigorous testing of the algorithm to ensure that any errors or limitations are removed from the calculation of the final offset polygon. When the algorithm performs accurately and with high reliability, it will be implemented into SDPS to update its offsetting function.

Additional work on the algorithm will include the inclusion of more user inputs to allow the user greater control over the final offset polygons that is created by the algorithm. These options are currently written as defaults within the algorithm. Whereas allowing for user control of these options, such as the chamfering of corners, would afford the user freedom to better control the output of the algorithm.

Appendix

Main.m

```
%This script will allow you to input the coordinates of a polygon
%and offset the polygon inward to account for edge effect offset by using the
%other functions called upon by the script.

%Order of functions that are performed:
%Main->segment->pinch->condense->testing->segment1->shaving->condense2

%---- Requested User Inputs -----
global v
global offset_dist
global o_error
o_error = 0.001

prompt_offset = 'What is the desired offset value?  ';
offset_dist = input(prompt_offset)

prompt_direction = 'Are the coordinates listed Clockwise(1) or Counter-Clockwise(2)? : ';
global CoordDirection
CoordDirection = input(prompt_direction,'s')
%---------------------------------

%----Outputs from called functions----
% Plot original
vertex_count = size(v,1);
f1 = (1:vertex_count);
patch('faces',f1,'vertices',v,'facealpha',0.01)
%-------------------------------------

% Segment
global v_new
v_new = segment(v);
% Plot segments
v_new_count = size(v_new,1);
f2 = (1:v_new_count);
%patch('faces',f2,'vertices',v_new,'facecolor','white')
%---------------------------------

% Pinch
global v_simple
v_simple = pinch(v_new);
% Plot pinch
simple_count = size(v_simple,1);
f3 = (1:simple_count);
%patch('faces',f3,'vertices',v_simple,'facealpha',0.01)
%-----------

% Condense
global v_condense
v_condense = condense(v_simple);
% Plot condense
v_condense_count = size(v_condense,1);
f4 = (1:v_condense_count);
%patch('faces',f4,'vertices',v_condense,'facealpha',0.01)
%------------

% Testing
global final_offset
final_offset = testing(v_condense)
% Plot final (before shave)
final_offset_count = size(final_offset,1);
f_final = (1:final_offset_count);
%patch('faces',f_final,'vertices',final_offset,'facealpha',0.01)
```

```
%------------

% Segment2
global v_new2
v_new2 = segment2(final_offset);
% Plot segment2
segment2_count = size(v_new2,1);
f_segment2 = (1:segment2_count);
%patch('faces',f_segment2,'vertices',v_new2,'facealpha',0.01)
%------------

% Shaving
global shave
shave = shaving(v_new2);
% Plot final (after shave)
shave_count = size(shave,1);
f_final_2 = (1:shave_count);
%patch('faces',f_final_2,'vertices',final,'facealpha',0.01)

% Condense2
offset = condense2(shave);
% Plot final (after shave)
offset_count = size(offset,1);
f_offset2 = (1:offset_count);
patch('faces',f_offset2,'vertices',offset,'facealpha',0.01)
%-------------------------------------------------------------------------
```

Segment.m

```
%This function splits the segments up into smaller segments to improve
%accuracy of the "pinch" function.
function [v_new] = segment(v)

vertex_count = size(v,1);
global offset_dist
local_offset_dist = offset_dist/1;
%----------------------------

count = 1;

% First coordinate through next to last coordinate
for i = 1:vertex_count-1

    % Original edge length
    dist = sqrt( ( v(i+1,1)-v(i,1) )^2 + ( v(i+1,2)-v(i,2) )^2 );

    % The first vertex of an edge (vertex 0)
    % Also considered the last vertex of the previous edge
    v_new(count,1) = v(i,1);
    v_new(count,2) = v(i,2);
    count = count + 1;

    if dist > local_offset_dist
        % Rounds up to determines number of segments to be made so that each segment is
        % less than "local offset dist" in length
        numb_seg = ceil( dist / local_offset_dist );
        % the length given to each new segment made
        seg_length = dist/numb_seg;
        % The number of new vertices to be added to the coordinate list.
        % The last vertex of the edge isn't MADE here, it is saved during
        % the next iteration of "i" as the vertex 0 for that edge.
        seg_created = numb_seg - 1;

        for j = 1:seg_created
            % Horizontal Line -------------------
            if v(i+1,2) - v(i,2) == 0
                % x2 > x1
                if v(i+1,1) - v(i,1) > 0
                    x_new = v(i,1) + (seg_length*j);
                    y_new = v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;
                % x2 < x1
                elseif v(i+1,1) - v(i,1) < 0
                    x_new = v(i,1) - (seg_length*j);
                    y_new = v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;
                end
                %-----------------------------------

            % Vertical Line --------------------
            elseif v(i+1,1) - v(i,1) == 0
                % y2 > y1
                if v(i+1,2) - v(i,2) > 0
                    x_new = v(i,1);
                    y_new = v(i,2) + (seg_length*j);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;

                    % y2 < y1
```

```
                elseif v(i+1,2) - v(i,2) < 0
                    x_new = v(i,1);
                    y_new = v(i,2) - (seg_length*j);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;

                end
                %-----------------------------------

            % Diagonal Lines -------------------
            % x2 > x1
            elseif v(i+1,1) - v(i,1) > 0
                % y2 > y1
                if v(i+1,2) - v(i,2) > 0
                    m = (v(i+1,2) - v(i,2)) / (v(i+1,1) - v(i,1));
                    x_new = v(i,1) + ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-v(i,1)) + v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif v(i+1,2) - v(i,2) < 0
                    m = (v(i+1,2) - v(i,2)) / (v(i+1,1) - v(i,1));
                    x_new = v(i,1) + ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-v(i,1)) + v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;
                end

            % x2 < x1
            elseif v(i+1,1) - v(i,1) < 0
                % y2 > y1
                if v(i+1,2) - v(i,2) > 0
                    m = (v(i+1,2) - v(i,2)) / (v(i+1,1) - v(i,1));
                    x_new = v(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-v(i,1)) + v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif v(i+1,2) - v(i,2) < 0
                    m = (v(i+1,2) - v(i,2)) / (v(i+1,1) - v(i,1));
                    x_new = v(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-v(i,1)) + v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;
                end
            end
        end
    end
end

% Last coordinate to first coordinate
for i = vertex_count

    dist = sqrt( ( v(1,1)-v(i,1) )^2 + ( v(1,2)-v(i,2) )^2 );
    v_new(count,1) = v(i,1);
    v_new(count,2) = v(i,2);
    count = count + 1;

    if dist > offset_dist
        % Rounds up to determines number of segments to be made so that each segment is
        % less than "local offset dist" in length
        numb_seg = ceil( dist / local_offset_dist );
```

```
% the length given to each new segment made
seg_length = dist/numb_seg;

% The number of new vertices to be added to the coordinate list.
% The last vertex of the edge isn't MADE here, it is saved during
% the next iteration of "i" as the vertex 0 for that edge.
seg_created = numb_seg - 1;

for j = 1:seg_created

    % Horizontal Line -------------------
    if v(1,2) - v(i,2) == 0
        % x2 > x1
        if v(1,1) - v(i,1) > 0
            x_new = v(i,1) + (seg_length*j);
            y_new = v(i,2);
            v_new(count,1) = x_new;
            v_new(count,2) = y_new;
            count = count + 1;

        % x2 < x1
        elseif v(1,1) - v(i,1) < 0
            x_new = v(i,1) - (seg_length*j);
            y_new = v(i,2);
            v_new(count,1) = x_new;
            v_new(count,2) = y_new;
            count = count + 1;
        end
        %-------------------------------------

    % Vertical Line --------------------
    elseif v(1,1) - v(i,1) == 0
        % y2 > y1
        if v(1,2) - v(i,2) > 0
            x_new = v(i,1);
            y_new = v(i,2) + (seg_length*j);
            v_new(count,1) = x_new;
            v_new(count,2) = y_new;
            count = count + 1;

        % y2 < y1
        elseif v(1,2) - v(i,2) < 0
            x_new = v(i,1);
            y_new = v(i,2) - (seg_length*j);
            v_new(count,1) = x_new;
            v_new(count,2) = y_new;
            count = count + 1;
        end
        %-------------------------------------

    % Diagonal Lines -------------------

    % x2 > x1
    elseif v(1,1) - v(i,1) > 0
        % y2 > y1
        if v(1,2) - v(i,2) > 0
            m = (v(1,2) - v(i,2)) / (v(1,1) - v(i,1));
            x_new = v(i,1) + ((seg_length*j)/sqrt(1+m^2));
            y_new = m*(x_new-v(i,1)) + v(i,2);
            v_new(count,1) = x_new;
            v_new(count,2) = y_new;
            count = count + 1;

        % y2 < y1
        elseif v(1,2) - v(i,2) < 0
            m = (v(1,2) - v(i,2)) / (v(1,1) - v(i,1));
            x_new = v(i,1) + ((seg_length*j)/sqrt(1+m^2));
            y_new = m*(x_new-v(i,1)) + v(i,2);
```

```
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;
                end

            % x2 < x1
            elseif v(1,1) - v(i,1) < 0
                % y2 > y1
                if v(1+1,2) - v(i,2) < 0
                    m = (v(1,2) - v(i,2)) / (v(1,1) - v(i,1));
                    x_new = v(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-v(i,1)) + v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif v(1,2) - v(i,2) < 0
                    m = (v(1,2) - v(i,2)) / (v(1,1) - v(i,1));
                    x_new = v(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-v(i,1)) + v(i,2);
                    v_new(count,1) = x_new;
                    v_new(count,2) = y_new;
                    count = count + 1;
                end
            end
        end
    end
end
v_new;

% function end
end
```

Pinch.m

```
%This function removes edges that will contribute to the creation of
%negative space based on the offset value chosen by the user.
function[v_simple] = pinch(v_new)

%For running using "main"
%global v_new
v_new_count = size(v_new,1);

global offset_dist
global o_error
global CoordDirection

% Data matrix column identification
% column 1 - Marked for deletion ( marked with "100" )
% column 2 - x coord
% column 3 - y coord
% column 4 - IEN.x coord
% column 5 - IEN.y coord
% column 6 - mid.x coord
% column 7 - mid.y coord
% column 8 - Intersect.x coord
% column 9 - Intersect.y coord
% column 10 - distance from intersect to 1-2 or 2-3 midpoint

data = zeros([v_new_count+2,10]);

% Input coordinates placed into columns 2 (x) and 3 (y).
for i = 1:v_new_count
    data(i,2) = v_new(i,1);
    data(i,3) = v_new(i,2);
end

% Next to last row receives coordinates of the first coordinate point.
data(v_new_count+1,2) = v_new(1,1);
data(v_new_count+1,3) = v_new(1,2);
% Last row receives the coordinates of the second coordinate point
data(v_new_count+2,2) = v_new(2,1);
data(v_new_count+2,3) = v_new(2,2);
%----------------------------------------

iteration = 1;
continue_loop = 1;
start1 = 1;

%---- Begin Iteration WHILE Loop ----
while continue_loop == 1
    iteration;

    %---- Selection of 3 coordinates for testing ----
    coord1 = 0;
    coord2 = 0;
    coord3 = 0;

    %---- Selection of first ----
    for i = start1:v_new_count
        if coord1 == 0
            if data(i,1) == 0
                first = i;
                % Stops the coord1 loop because a coordinate has been
                % chosen.
                coord1 = 1;
                start2 = first+1;
            end
        end
    end
```

```
    %----------------------------

    %---- Selection of Second ----
    if coord1 == 1
        % If "first" is "1" then it can't use v_new_count+1 or +2 as "second" or "third".
        if first == 1
            for i = start2:v_new_count
                if coord2 == 0
                    if data(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end

        % If "first" is "2" then it can't use v_new_count+2 as "second" or "third".
        elseif first == 2
            for i = start2:v_new_count+1
                if coord2 == 0
                    if data(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end

        % If "first" is anything except 1 or 2, then it is acceptable to use
v_new_count+1 and +2 as "second" or "third".
        else
            for i = start2:v_new_count+2
                if coord2 == 0
                    if data(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end
        end
    end

    %---- Selection of Third ----
    if coord1 == 1
        if coord2 == 1
            % If "first" is "1" then it can't use v_new_count+1 or +2 as "second" or
"third".
            if first == 1
                for i = start3:v_new_count
                    if coord3 == 0
                        if data(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end

            % If "first" is "2" then it can't use v_new_count+2 as "second" or "third".
            elseif first == 2
                for i = start3:v_new_count+1
                    if coord3 == 0
                        if data(i,1) == 0
```

```
                            third = i;
                            coord3 = 1;
                        end
                    end
                end

            % If "first" is anything except 1 or 2, then it is acceptable to use
v_new_count+1 and +2 as "second" or "third".
            else
                for i = start3:v_new_count+2
                    if coord3 == 0
                        if data(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end
            end
        end
    end

    %---- Only calculates this section if three coordinates were chosen ----
    if coord1 == 1
        if coord2 == 1
            if coord3 == 1
                % If dd=0 then the lines are parallel. Don't run next steps
                % for parallel lines.
                dd = ( (data(third,3)-data(second,3))*(data(second,2)-data(first,2)) ) -
( (data(third,2)-data(second,2))*(data(second,3)-data(first,3)) );

           %---- Calculation of IEN for the chosen points ----
                % If the 2 lines are parallel
                if abs(dd) < o_error
                    %disp('Lines in parallel');
                    iteration = iteration + 1;
                    start1 = start1 + 1;

                %if dd ~= 0 (NOT in parallel)
                elseif abs(dd) >= o_error
                    %---- First to Second ----
                    dx_12 = data(second,2) - data(first,2);
                    dy_12 = data(second,3) - data(first,3);
                    edgelength_12 = (dx_12*dx_12 + dy_12*dy_12)^(.5);

                    % Clockwise Coordinates
                    if CoordDirection == '1'
                        x_in_12 = dy_12/edgelength_12;
                        y_in_12 = -dx_12/edgelength_12;
                        data(first,4) = x_in_12;
                        data(first,5) = y_in_12;

                    % Counter-Clockwise Coordinates
                    elseif CoordDirection == '2'
                        x_in_12 = -dy_12/edgelength_12;
                        y_in_12 = dx_12/edgelength_12;
                        data(first,4) = x_in_12;
                        data(first,5) = y_in_12;
                    end

                    %---- Second to Third ----
                    dx_23 = data(third,2) - data(second,2);
                    dy_23 = data(third,3) - data(second,3);
                    edgelength_23 = (dx_23*dx_23 + dy_23*dy_23)^(.5);

                    % Clockwise Coordinates
                    if CoordDirection == '1'
                        x_in_23 = dy_23/edgelength_23;
                        y_in_23 = -dx_23/edgelength_23;
                        data(second,4) = x_in_23; %x of unit normal
```

```
                    data(second,5) = y_in_23;

                % Counter-Clockwise Coordinates
                elseif CoordDirection == '2'
                    x_in_23 = -dy_23/edgelength_23;
                    y_in_23 = dx_23/edgelength_23;
                    data(second,4) = x_in_23; %x of unit normal
                    data(second,5) = y_in_23;
                end
                %-----------------------------------------------------

        %---- Calculation of Mid point for the chosen points ----

                %---- First to Second ----
                % x-coord
                data(first,6) = (data(first,2) + data(second,2))/2;
                % y-coord
                data(first,7) = (data(first,3) + data(second,3))/2;

                %---- Second to Third ----
                % x-coord
                data(second,6) = (data(second,2) + data(third,2))/2;
                % y-coord
                data(second,7) = (data(second,3) + data(third,3))/2;
        %-------------------------------------------------------



        %---- Calculation of Intersection between 1-2's Midpoint/IEN with 2-3's
        %---- Midpoint/IEN ----

                u = ((data(first,7)*data(second,4)) + (data(second,5)*data(second,6))
- (data(second,7)*data(second,4)) - (data(second,5)*data(first,6))) /
((data(first,4)*data(second,5))-(data(first,5)*data(second,4)));
                v = (data(first,6)+(data(first,4)*u) -
data(second,6))/data(second,4);

                % v is not calculated when the denominator is zero
                if data(second,4) == 0
                    v = 1;
                end

                % If u and v are both positive then the rays will intersect
                % (Before the edges are offset)
                if u > 0 && v > 0

                    %---- Determine Offset Coordinates ----
                    % Edge 1
                    % first vertex (point 1)
                    edge1_x1 = data(first,2) + (offset_dist*x_in_12);
                    edge1_y1 = data(first,3) + (offset_dist*y_in_12);
                    % second vertex (point 2)
                    edge1_x2 = data(first+1,2) + (offset_dist*x_in_12);
                    edge1_y2 = data(first+1,3) + (offset_dist*y_in_12);

                    % Edge 2
                    % first vertex (point 3)
                    edge2_x3 = data(second,2) + (offset_dist*x_in_23);
                    edge2_y3 = data(second,3) + (offset_dist*y_in_23);
                    %second vertex ( point 4)
                    edge2_x4 = data(second+1,2) + (offset_dist*x_in_23);
                    edge2_y4 = data(second+1,3) + (offset_dist*y_in_23);

                    %Calculate new midpoints for the edges that are offset
                    % and use the same IENs already calculated to find the
                    % u and v for the new positions.

                    %---- Calculation of Mid points of offset edges ----
                    %---- First to Second ----
```

```
                        % x-coord
                        offset1_mid_x = (edge1_x1 + edge1_x2)/2;
                        % y-coord
                        offset1_mid_y = (edge1_y1 + edge1_y2)/2;
                        %---- Second to Third ----
                        % x-coord
                        offset2_mid_x = (edge2_x3 + edge2_x4)/2;
                        % y-coord
                        offset2_mid_y = (edge2_y3 + edge2_y4)/2;
                        %-------------------------------------------------------

                        %---- Test for Intersection between 1-2's Midpoint/IEN with 2-3's
                        %---- Midpoint/IEN ----
                        offset_u = ((offset1_mid_y*x_in_23) + (y_in_23*offset2_mid_x) -
(offset2_mid_y*x_in_23) - (y_in_23*offset1_mid_x)) / ((x_in_12*y_in_23) -
(y_in_12*x_in_23));
                        offset_v = (offset1_mid_x + (x_in_12*offset_u) - offset2_mid_x) /
x_in_23;

                        % v is not calculated when the denominator is zero
                        if x_in_23 == 0
                            offset_v = 1;
                        end

                        % They won't intersect because the offset was too large and will
cause an internal loop if not deleted(bad)
                        if offset_u <= 0 || offset_v <= 0
                            data(second,1) = 100;
                            iteration = iteration + 1;
                            disp('Offset value chosen was too large for the test
coordinates')

                        %If u and v are both positive then the rays will STILL intersect
and now will test the distance (good)
                        % (After the edges are offset)
                        elseif offset_u > 0 && offset_v > 0

                            %disp('intersection happens in ray directions')
                            % Intersection Point
                            data(first,8) = data(first,6) + (data(first,4)*u);
                            data(first,9) = data(first,7) + (data(first,5)*u);

                            % Calculation of distance from intersect to 1-2 midpoint and
2-3 midpoint
                            data(first,10) = sqrt((data(first,8)-data(first,6))^2 +
(data(first,9)-data(first,7))^2);
                            data(second,10) = sqrt((data(first,8)-data(second,6))^2 +
(data(first,9)-data(second,7))^2);

                            % Marking coordinates for deletion
                            if abs(data(first,10) - offset_dist) <= o_error ||
abs(data(second,10) - offset_dist) <= o_error

                                % Second point is marked for deletion
                                data(second,1) = 100;
                                iteration = iteration + 1;
                                data;

                            % Nothing Deleted
                            else  %if data(first,10) >= offset_dist && data(second,10) >=
offset_dist
                                % If nothing is deleted then it is time to move the
"first" coordinate to the next available option.
                                start1 = start1 + 1;
                                disp('intersection point is far enough away from the
edges that created it')
                                iteration = iteration + 1;
                                data;
                            end
```

```
                    end


                else %If u and/or v is less than 0 then there is no intersection.
                    start1 = start1 + 1;
                    iteration = iteration + 1;
                end
            end
        end
    end
    %-------------------------------------------------

    %---- Coordinates 1 or 2 marked for deletion if correspinding extension coordinates
are marked ----
    if data(v_new_count+1,1) == 100
        data(1,1) = 100;
    end

    if data(v_new_count+2,1) == 100
        data(2,1) = 100;
    end
    %-------------------------------------------------------------------------

    %---- Stop While loop conditions ----
    %                Need coord2 or coord3 ?
    if coord1 == 0 || coord2 == 0 || coord3 ==0
        continue_loop = 0;
    end
    %-----------------------------------

% End "ITERATION" loop
end

%---- Simplifying Coordinate list ----
temp = v_new;

for i = v_new_count:-1:1
    if data(i,1) == 100
        temp(i,:) = [];
    end
end
%-------------------------------------

data;

% End Function
end
```

Condense.m

```
% This subroutine will condense line segments in series that are parallel
% to each other into a single line segment
function [v_condense] = condense(v_simple)

simple_count = size(v_simple,1);
global o_error
% Testing
%v_simple = [3 0; 6 0; 9 0; 10 3; 10 6; 10 9; 7 10; 4 10; 1 10; 0 7; 0 4; 0 1]
%simple_count = size(v_simple,1)

%---- Create temp matrix for coordinates to be marked for deletion ----
temp = zeros([simple_count+2,3]);
for i = 1:simple_count
    temp(i,2) = v_simple(i,1);
    temp(i,3) = v_simple(i,2);
end

% Next to last row receives coordinates of the first coordinate point.
temp(simple_count+1,2) = v_simple(1,1);
temp(simple_count+1,3) = v_simple(1,2);
% Last row receives the coordinates of the second coordinate point
temp(simple_count+2,2) = v_simple(2,1);
temp(simple_count+2,3) = v_simple(2,2);
%----------------------------------------

%-----------------
iteration = 1;
continue_loop = 1;
start1 = 1;

%---- Begin Iteration WHILE Loop ----
while continue_loop == 1
    iteration;

    %---- Selection of 3 coordinates for testing ----
    coord1 = 0;
    coord2 = 0;
    coord3 = 0;
    %---- Selection of first ----
    for i = start1:simple_count
        if coord1 == 0
            if temp(i,1) == 0
                first = i;
                % Stops the coord1 loop because a coordinate has been
                % chosen.
                coord1 = 1;
                start2 = first+1;
            end
        end
    end
    %----------------------------

    %---- Selection of Second ----
    if coord1 == 1
        % If "first" is "1" then it can't use simple_count+1 or +2 as "second" or
"third".
        if first == 1
            for i = start2:simple_count
                if coord2 == 0
                    if temp(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
```

```
                    end
                end
            end

        % If "first" is "2" then it can't use simple_count+2 as "second" or "third".
        elseif first == 2
            for i = start2:simple_count+1
                if coord2 == 0
                    if temp(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end

        % If "first" is anything except 1 or 2, then it is acceptable to use
simple_count+1 and +2 as "second" or "third".
        else
            for i = start2:simple_count+2
                if coord2 == 0
                    if temp(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end
        end
    end

    %---- Selection of Third ----
    if coord1 == 1
        if coord2 == 1

            % If "first" is "1" then it can't use simple_count+1 or +2 as "second" or
"third".
            if first == 1
                for i = start3:simple_count
                    if coord3 == 0
                        if temp(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end

            % If "first" is "2" then it can't use simple_count+2 as "second" or "third".
            elseif first == 2
                for i = start3:simple_count+1
                    if coord3 == 0
                        if temp(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end

            % If "first" is anything except 1 or 2, then it is acceptable to use
simple_count+1 and +2 as "second" or "third".
            else
                for i = start3:simple_count+2
                    if coord3 == 0
                        if temp(i,1) == 0
                            third = i;
```

```
                            coord3 = 1;
                        end
                    end
                end
            end
        end
    end

    %---- Only calculates this section if three coordinates were chosen ----
    if coord1 == 1
        if coord2 == 1
            if coord3 == 1
                dd = ( (temp(third,3)-temp(second,3))*(temp(second,2)-temp(first,2)) ) -
( (temp(third,2)-temp(second,2))*(temp(second,3)-temp(first,3)) );
                % If dd=0 then the lines are parallel.
                % parallel lines not condensed here cause problems later.

                % Error allowance
                if abs(dd) < o_error
                    temp(second,1) = 100;
                    iteration = iteration + 1;

                % Not in parallel
                % Error allowance
                elseif abs(dd) >= o_error
                    start1 = start1 + 1;
                    iteration = iteration + 1;
                end
            end
        end
    end

    %---- Coordinates 1 or 2 marked for deletion if correspinding extension coordinates
are marked ----
    if temp(simple_count+1,1) == 100
        temp(1,1) = 100;
    end

    if temp(simple_count+2,1) == 100
        temp(2,1) = 100;
    end
    %-------------------------------------------------------------------------

    %---- Stop While loop conditions ----
    %                Need coord2 or coord3 ?
    if coord1 == 0 || coord2 == 0 || coord3 ==0
        continue_loop = 0;
    end
    %-----------------------------------

% While loop end
end

%---- Simplifying Coordinate list ----
temp2 = v_simple;

for i = simple_count:-1:1
    if temp(i,1) == 100
        temp2(i,:) = [];
    end
end
%---------------------------------------
v_condense = temp2;
% Function end
end
```

Testing.m

```
function[final_offset] = testing(v_condense)

% Chooses 3 coordinates (2 edges) by ignoring any edges that were
% determined to create bad vertices when the intersect of the offsets was
% calculate (this step performed after vertices are chosen).

% This function determines the IEN of both edges, offsets the vertices (the
% shared vertex is offset for each IEN), and calculates the intersection
% point. If the intersection is too close to one of the original edges then
% it is NOT saved and the function determines the next set of vertices to
% test.

v_condense_count = size(v_condense,1);
global offset_dist
global o_error
global CoordDirection

% Data matrix column identification
% column 1 - Marked for deletion ( marked with "100" )
% column 2 - x coord
% column 3 - y coord

data = zeros([v_condense_count+2,3]);

% Input coordinates placed into columns 2 (x) and 3 (y).
for i = 1:v_condense_count
    data(i,2) = v_condense(i,1);
    data(i,3) = v_condense(i,2);
end
% Next to last row receives coordinates of the first coordinate point.
data(v_condense_count+1,2) = v_condense(1,1);
data(v_condense_count+1,3) = v_condense(1,2);
% Last row receives the coordinates of the second coordinate point
data(v_condense_count+2,2) = v_condense(2,1);
data(v_condense_count+2,3) = v_condense(2,2);

% for collecting correct offset coordinates
count = 1;
iteration = 1;
continue_loop = 1;
start1 = 1;

%---- Begin Iteration WHILE Loop ----
while continue_loop == 1
    iteration;
    %---- Selection of 3 coordinates for testing ----
    coord1 = 0;
    coord2 = 0;
    %---- Selection of first ----
    for i = start1:v_condense_count
        if coord1 == 0
            if data(i,1) == 0
                first = i
                % Stops the coord1 loop because a coordinate has been
                % chosen.
                coord1 = 1;
                start2 = first+1;
            end
        end
    end
    %----------------------------

    %---- Selection of Second ----
    if coord1 == 1
        % If "first" is the first vertex then "second" cannot be v_condense_count+1
```

```
        if first == 1
            for i = start2:v_condense_count
                if coord2 == 0
                    if data(i,1) == 0
                        second = i
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                    end
                end
            end

        % If "first" is not the last vertex then "second" can be anything
        else
            for i = start2:v_condense_count+1
                if coord2 == 0
                    if data(i,1) == 0
                        second = i
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                    end
                end
            end
        end
    end
%-------------------------

%---- Only calculates this section if two coordinates were chosen ----
    if coord1 == 1
        if coord2 == 1
            %IEN gives the direction of the normal the points towards the center
            %of the polygon, OEN gives the direction away from the center.
            %Move the edge of the polygon in the direction of the IEN to
            %offset the polygon inside of itself.

            %Moves the vertices of each line segment by the IENoffset value. Vertices
            %get moved twice each to compensate for vertices that are shared by 2
            %segments.

            %---- Calculate Parameters for IEN ----
                %---- For Edge 1 ----
            dx_1 = data(first+1,2) - data(first,2);
            dy_1 = data(first+1,3) - data(first,3);
            edgelength_1 = (dx_1*dx_1 + dy_1*dy_1)^(.5);

                %---- For Edge 2 ----
            dx_2 = data(second+1,2) - data(second,2);
            dy_2 = data(second+1,3) - data(second,3);
            edgelength_2 = (dx_2*dx_2 + dy_2*dy_2)^(.5);

            %--------------------------------------

            %---- Inward Edge Normal ----
                %---- For Counter Clockwise Coordinates ----
            if CoordDirection == '2'
                IEN_1_x = -dy_1/edgelength_1;
                IEN_1_y = dx_1/edgelength_1;
                IEN_2_x = -dy_2/edgelength_2;
                IEN_2_y = dx_2/edgelength_2;
                %---- For Clockwise Coordinates ----
            elseif CoordDirection == '1'
                IEN_1_x = dy_1/edgelength_1;
                IEN_1_y = -dx_1/edgelength_1;
                IEN_2_x = dy_2/edgelength_2;
                IEN_2_y = -dx_2/edgelength_2;
            end
            %---------------------------
```

```
            %---- Determine Offset Coordinates ----
            % Edge 1
                % first vertex (point 1)
                edge1_x1 = data(first,2) + (offset_dist*IEN_1_x);
                edge1_y1 = data(first,3) + (offset_dist*IEN_1_y);
                % second vertex (point 2)
                edge1_x2 = data(first+1,2) + (offset_dist*IEN_1_x);
                edge1_y2 = data(first+1,3) + (offset_dist*IEN_1_y);
            % Edge 2
                % first vertex (point 3)
                edge2_x3 = data(second,2) + (offset_dist*IEN_2_x);
                edge2_y3 = data(second,3) + (offset_dist*IEN_2_y);
                %second vertex ( point 4)
                edge2_x4 = data(second+1,2) + (offset_dist*IEN_2_x);
                edge2_y4 = data(second+1,3) + (offset_dist*IEN_2_y);
            %---------------------------------------

            %---- Calculate Intersection Point ----
            dd = ((edge2_y4-edge2_y3)*(edge1_x2-edge1_x1)) - ((edge2_x4-
edge2_x3)*(edge1_y2-edge1_y1));
            n_a = ((edge2_x4-edge2_x3)*(edge1_y1-edge2_y3)) - ((edge2_y4-
edge2_y3)*(edge1_x1-edge2_x3));
            u_a = n_a/dd;

            %intersection coordinates
            intersect_x = edge1_x1 + (u_a * (edge1_x2-edge1_x1));
            intersect_y = edge1_y1 + (u_a * (edge1_y2-edge1_y1));
            %---------------------------------------

            distance_fail = 0;
            %---- Distance to Original Vertices Test ----
            % If the intersection is too close to any original vertex then the proceeding
distance
            %tests are unneccessary for that intersection,
            vertex_dist_count = 0;
            for j = 1:v_condense_count
                dist_vertex = sqrt((intersect_x-v_condense(j,1))^2 + (intersect_y-
v_condense(j,2))^2);
                if dist_vertex < offset_dist
                    % Counts how many original vertices the intersection it
                    % too close to.
                    vertex_dist_count = vertex_dist_count + 1;
                    j;
                end
            end

            % If the intersection is too close to one or more original vertices
            % then the intersection point is bad and it goes to the next set of points.
            if vertex_dist_count > 0
                iteration = iteration + 1;
                distance_fail = 1;
                start1 = start1 + 1;
                disp('too close to original vertex and incorrect');
            end
            %----------------------------------------

            %---- First Distance Test ----
            % checks the shortest distance between the intersection point against all
edges of data
            % Only runs the calculations if a point has not failed a test
            if distance_fail == 0
                check1 = zeros([v_condense_count,1]);
                dist = zeros([v_condense_count,1]);
                for j = 1:v_condense_count
                    % coefficients
                    a = data(j+1,3) - data(j,3);
                    b = data(j+1,2) - data(j,2);
                    c = data(j+1,2)*data(j,3) - data(j+1,3)*data(j,2);
                    %r
```

```
                    dist(j,1) = abs((a*intersect_x) - (b*intersect_y) + c) / sqrt(a^2 +
b^2);
                end

                % Tests for failure of first distance test (added an error allowance)
                for j = 1:v_condense_count
                    if dist(j,1) < offset_dist-o_error
                        check1(j,1) = 100;
                        disp('Distance Test 1: too close to original edge, sent to
Distance Test 2');
                    end
                end
                check1;
            end
            %-----------------------------

            %---- Second Distance Test ----
            %Save coordinate from a false positive.

            % Only runs the calculations if a point has not failed a test
            if distance_fail == 0
                check2 = check1;

                for j = 1:v_condense_count
                    if check1(j,1) == 100
                        % horizontal lines have a zero in the denominator when taking the
inverse slope
                        % y1 = y2
                        if data(j,3) == data(j+1,3)
                            if intersect_x < data(j,2) || intersect_x > data(j+1,2)
                                check2(j,1) = 0;
                            end
                        end

                        if data(j,3) ~= data(j+1,3)
                            inverse_m = (-1)*(data(j+1,2)-data(j,2))/(data(j+1,3)-
data(j,3));
                            b = intersect_y - (inverse_m * intersect_x);
                            dd = ((intersect_y-b)*(data(j+1,2)-data(j,2))) -
((intersect_x-0)*(data(j+1,3)-data(j,3)));
                            na = ((intersect_x-0)*(data(j,3)-b)) - ((intersect_y-
b)*(data(j,2)-0));
                            nb = ((data(j+1,2)-data(j,2))*(data(j,3)-b)) - ((data(j+1,3)-
data(j,3))*(data(j,2)-0));
                            ua = na/dd;
                            ub = nb/dd;

                            %if 0<=ua && ua<=1 && 0<=ub && ub<=1;
                            x_new = data(j,2) + (ua * (data(j+1,2) - data(j,2)));
                            %y_new = data(j,3) + (ua * (data(j+1,3) - data(j,3)))

                            %Diagonal lines can be accurately tested with only a single
                            %comparison, either x or y. Need 2 methods of x comparison
                            %to account for x1<x2 or x1>x2.

                            %Y comparisons are required for vertical lines. Need -1
                            %methods as mentioned previously. y1<y2 and y1>y2.
                            %X comparisons are required for horizontal lines

                            % x1 < x2
                            if  data(j,2) < data(j+1,2)
                                if x_new < data(j,2) || x_new > data(j+1,2)
                                    check2(j,1) = 0;
                                end
                                % x1 > x2
                            elseif data(j,2) >= data(j+1,2)
                                if x_new <= data(j+1,2) || x_new >= data(j,2)
                                    check2(j,1) = 0;
                                end
```

```
                        end
                    end
                end
            end

            % adds the column of the check2 matrix
            check_sum2 = sum(check2);

            % The intersection is too close to one or more v_condense edges
            if check_sum2 > 0
                iteration = iteration + 1;
                start1 = start1 + 1;
                disp('Distance Test 2: too close to original edge and incorrect');
            end
        end

        %---- Inside or Outside Original Polygon Check ----

        % Determines whether the intersectionpoint is inside the original
        % polygon. If it is outside then it is an incorrect point. If
        % it is inside then it is saved as a final offset polygon
        % vertex.

        % This test will count the number of times that a horizontal
        % line through the intersection point will intersect with an
        % original polygon edge. If the total number of instersection
        % is an odd number, the intersection point is considered inside
        % the original polygon.

        % Only calculates this step if the Intersection was not too
        % close to any original edge.
        % Only runs the calculations if a point has not failed a test.
        if distance_fail == 0
            if check_sum2 == 0
                intersection_count = 0;
                % Creates a second point to pair with the intersection
                % point to create a horizontal line segment.
                test = [intersect_x + 1, intersect_y];
                for j = 1:v_condense_count
                    % Checks if one or both edge x-coordinates is greater than or
equal to the x-coord of the intersection point.
                    % This way it reduces the number of calculations made if there is
NO chance that an
                    % intersection TO THE RIGHT of the intersection point can occur.
                    if data(j,2) >= intersect_x || data(j+1,2) >= intersect_x

                        % if original edge is vertical
                        % x1 = x2
                        if data(j,2) == data(j+1,2)
                            % If the y-coord of the intersection point is inside the
y
                            % limits of the original edge then its a good
intersection

                            % y1 < y2
                            if data(j,3) < data(j+1,3)
                                if intersect_y > data(j,3) && intersect_y <
data(j+1,3)
                                    intersection_count = intersection_count + 1;
                                end

                                % y1 > y2
                            elseif data(j,3) > data(j+1,3)
                                if intersect_y < data(j,3) && intersect_y >
data(j+1,3)
                                    intersection_count = intersection_count + 1;
                                end
                            end
```

```
                            % if original edge is NOT vertical
                            elseif data(j,2) ~= data(j+1,2)
                                % If the original edge is NOT horizontal (horizontal
lines don't
                                % contribute to the intersection count)
                                if data(j,3) ~= data(j+1,3)
                                    % point 1 = edge vertex j
                                    % point 2 = edge vertex j+1
                                    % point 3 = intersection point
                                    % point 4 = test point
                                    dd = ((test(1,2)-intersect_y)*(data(j+1,2)-
data(j,2))) - ((test(1,1)-intersect_x)*(data(j+1,3)-data(j,3)));
                                    na = ((test(1,1)-intersect_x)*(data(j,3)-
intersect_y)) - ((test(1,2)-intersect_y)*(data(j,2)-intersect_x));
                                    %nb = ((data(j+1,2)-data(j,2))*(data(j,3)-
intersect_y)) - ((data(j+1,3)-data(j,3))*(data(j,2)-intersect_x));
                                    ua = na/dd;
                                    %ub = nb/dd;
                                    %if 0<=ua && ua<=1 && 0<=ub && ub<=1;
                                    inter_x = data(j,2) + (ua * (data(j+1,2) -
data(j,2)));
                                    inter_y = data(j,3) + (ua * (data(j+1,3) -
data(j,3)));

                                    %Diagonal lines can be accurately tested with only a
single
                                    %comparison, either x or y. Need 2 methods of x
comparison
                                    %to account for x1<x2 or x1>x2.

                                    % Checks if inter_x is to the right of the calculated
intersection point.
                                    % This way it only calculates intersections that
occur to the right of the intersection point being tested.
                                    if inter_x >= intersect_x
                                        % x1 < x2
                                        if  data(j,2) < data(j+1,2)
                                            if inter_x > data(j,2) && inter_x <
data(j+1,2)
                                                1;
                                                inter_x;
                                                inter_y;
                                                x1 = data(j,2);
                                                y1 = data(j,3);
                                                x2 = data(j+1,2);
                                                y2 = data(j+1,3);
                                                intersection_count = intersection_count +
1;
                                            end

                                        % x1 > x2
                                        elseif data(j,2) > data(j+1,2)
                                            if inter_x < data(j,2) && inter_x >
data(j+1,2)
                                                2;
                                                inter_x;
                                                inter_y;
                                                x1 = data(j,2);
                                                y1 = data(j,3);
                                                x2 = data(j+1,2);
                                                y2 = data(j+1,3);
                                                intersection_count = intersection_count +
1;
                                            end
                                        end
                                    end
                                end
                            end
                        end
```

```
                    end
                    final_intersection_count = intersection_count;

                    % Detemines if intersection is inside original polygon
                    % returns 1 if odd and 0 if even
                    odd = mod(final_intersection_count,2);

                    % odd number of intersections (good)
                    % Intersection point moves on to the point shaving test
                    if odd ~= 0
                        disp('Intersection Point is inside original polygon')
                    % even number of intersections (bad)
                    elseif odd == 0
                        iteration = iteration + 1;
                        start1 = start1 + 1;
                        disp('Intersection Point outside original polygon and incorrect')
                    end
                end
            end
            %-------------------------------------------------

            %---- Point Shaving Test ----
            % Determines if the intersection point is too far from the
            % vertex point and if so, will calculate 2 coordinates to
            % replace the intersection point to act as a chamfer on the
            % point

            % Calculates u and v first to determine if the selected
            % coordinate form an arrowhead before doing any calculations
            if distance_fail == 0
                if odd ~= 0
                    %---- Calculation of Mid point for the chosen points ----

                    %---- First to Second ----
                    % x-coord
                    mid_1_x = (data(first,2) + data(first+1,2))/2;
                    % y-coord
                    mid_1_y = (data(first,3) + data(first+1,3))/2;

                    %---- Second to Third ----
                    % x-coord
                    mid_2_x = (data(second,2) + data(second+1,2))/2;
                    % y-coord
                    mid_2_y = (data(second,3) + data(second+1,3))/2;
                    %-----------------------------------------------------------
                    u = ( (mid_1_y*IEN_2_x) + (IEN_2_y*mid_2_x) - (mid_2_y*IEN_2_x) -
(IEN_2_y*mid_1_x) ) / ( (IEN_1_x*IEN_2_y) - (IEN_1_y*IEN_2_x) ) ;
                    v = ( mid_1_x + (IEN_1_x*u) - mid_2_x ) / IEN_2_x;

                    % v is not calculated when the denominator is zero
                    if IEN_2_x == 0
                        v = 1;
                    end

                    % If u and v are both positive then the rays will intersect
                    % This portion of the code only performs calculations
                    % for edges with rays that would NOT intersect (arrowheads)
                    if u <= 0 || v <= 0
                        disp('u or v negative, next step')

                        %test distance from arrowhead to the intersection
                        %point
                        distance_inter_point = sqrt( ( intersect_x - data(second,2) )^2 +
( intersect_y - data(second,3) )^2 );

                        if distance_inter_point > offset_dist*1.5
                            disp('intersection too far, next step')
                            slope_inter_second = (intersect_y - data(second,3)) /
(intersect_x - data(second,2));
```

```
                            c = intersect_y - (slope_inter_second * intersect_x);

                            % calculate intersection between this line
                            % and a circle with center at second with
                            % radius of offset_dist
                            A = slope_inter_second^2 + 1;
                            B = 2*((slope_inter_second*c) -
(slope_inter_second*data(second,3)) - data(second,2));
                            C = data(second,3)^2 - offset_dist^2 + data(second,2)^2 -
(2*c*data(second,3)) + c^2;

                            %calculates the x-coord of the intersect
                            %with the quadratic formula
                            quad1 = (-B - sqrt(B^2 - (4*A*C))) / (2*A);
                            quad2 = (-B + sqrt(B^2 - (4*A*C))) / (2*A);

                            % Identifies orientation and chooses the
                            % quad result that lies between intersect
                            % and second
                            if intersect_x >= data(second,2)
                                if intersect_x >= quad1 && quad1 >= data(second,2)
                                    inter_x = quad1;
                                elseif intersect_x >= quad2 && quad2 >= data(second,2)
                                    inter_x = quad2;
                                end

                            elseif intersect_x <= data(second,2)
                                if intersect_x <= quad1 && quad1 <= data(second,2)
                                    inter_x = quad1;
                                elseif intersect_x <= quad2 && quad2 <= data(second,2)
                                    inter_x = quad2;
                                end
                            end

                            inter_y = (slope_inter_second*inter_x) + c;

                            % Calculates the line equation of the line
                            % perpendicular to the previous line
                            % that will help calculate the two
                            % coordinates to replace the intersection.
                            perp_slope = -1/slope_inter_second;
                            perp_c = inter_y - (perp_slope * inter_x);

                            % Calculates the intersection points
                            % between this new line and the two offset
                            % edges to find the coordinates that will
                            % replace the intersection point and saves
                            % them to the final coordinate list

                            %---- First Coordinate ----
                            slope_edge_1 = (edge1_y2 - edge1_y1) / (edge1_x2 - edge1_x1);
                            edge1_c = edge1_y1 - (slope_edge_1 * edge1_x1);
                            new_inter1_x = (perp_c - edge1_c) / (slope_edge_1 -
perp_slope);
                            new_inter1_y = slope_edge_1 * (new_inter1_x) + edge1_c;

                            offset(count,1) = new_inter1_x;
                            offset(count,2) = new_inter1_y;
                            count = count + 1;

                                %---- Second Coordinate ----
                                slope_edge_2 = (edge2_y4 - edge2_y3) / (edge2_x4 -
edge2_x3);
                                edge2_c = edge2_y3 - (slope_edge_2 * edge2_x3);
                                new_inter2_x = (perp_c - edge2_c) / (slope_edge_2 -
perp_slope);
                                new_inter2_y = slope_edge_2 * (new_inter2_x) + edge2_c;
                                offset(count,1) = new_inter2_x;
                                offset(count,2) = new_inter2_y;
```

```
                                count = count + 1;

                                % This is the last calculation of the
                                % iteration so start and iteration get 1
                                % added to them
                                start1 = start1 + 1;
                                iteration = iteration + 1;

                            % Not too far from original source vertex, coordinate is saved
                            else
                                offset(count,1) = intersect_x;
                                offset(count,2) = intersect_y;
                                count = count + 1;
                                start1 = start1 + 1;
                                iteration = iteration + 1;
                            end

                        % If u and v are both positive then the rays will intersect
                        % Then the intersection point calculated is
                        % correct and saved.
                        elseif u > 0 && v > 0
                            count;
                            offset(count,1) = intersect_x;
                            offset(count,2) = intersect_y
                            count = count + 1;
                            start1 = start1 + 1;
                            iteration = iteration + 1;
                        end
                    end
                end
                %-----------------------------------------------------------------
            end
        end

        %---- Stop While loop conditions ----
        if coord1 == 0 || coord2 == 0
            continue_loop = 0;
        end
        %-------------------------------------

    % end iteration loop
    end

    %---- Repeated Coordinate Test -----
    %i = base coordinate
    %j= compared coordinate
    offset_count = size(offset,1);
    new_offset = zeros([offset_count,offset_count]);
    for i = offset_count:-1:2
        for j = i-1:-1:1
            if abs((offset(i,1)-offset(j,1))/offset(i,1))*100 <= 0.000001
                if abs((offset(i,2)-offset(j,2))/offset(i,2))*100 <= 0.000001
                    new_offset(j,i) = 1000;
                    i;
                    j;
                end
            else
                new_offset(j,i) = 0;
            end
        end
    end

    %----Simplifies "new_offset" matrix for next elimination step----
    new_offset2 = sum(new_offset);
    %---------------------------------------------------------

    %----Vertex Elimination----
    count_final = 1;
    for i = 1:offset_count
```

```
    if new_offset2(1,i) == 0
        clean_offset(count_final,1) = offset(i,1);
        clean_offset(count_final,2) = offset(i,2);
        count_final = count_final + 1;
    end
end
%----------------------------------------

final_offset = clean_offset;

% function end
end
```

Segment2.m

```
function [v_new2] = segment2(final_offset)

final_offset_count = size(final_offset,1);
global offset_dist
local_offset_dist = offset_dist/1;
%----------------------------

count = 1;
% First coordinate through next to last coordinate
for i = 1:final_offset_count-1
    % Original edge length
    dist = sqrt( ( final_offset(i+1,1)-final_offset(i,1) )^2 + ( final_offset(i+1,2)-
final_offset(i,2) )^2 );
    % The first vertex of an edge (vertex 0)
    % Also considered the last vertex of the previous edge
    v_new2(count,1) = final_offset(i,1);
    v_new2(count,2) = final_offset(i,2);
    count = count + 1;
    if dist > local_offset_dist
        % Rounds up to determines number of segments to be made so that each segment is
        % less than "local offset dist" in length
        numb_seg = ceil( dist / local_offset_dist );
        % the length given to each new segment made
        seg_length = dist/numb_seg;
        % The number of new vertices to be added to the coordinate list.
        % The last vertex of the edge isn't MADE here, it is saved during
        % the next iteration of "i" as the vertex 0 for that edge.
        seg_created = numb_seg - 1;

        for j = 1:seg_created
            % Horizontal Line -------------------
            if final_offset(i+1,2) - final_offset(i,2) == 0
                % x2 > x1
                if final_offset(i+1,1) - final_offset(i,1) > 0
                    x_new = final_offset(i,1) + (seg_length*j);
                    y_new = final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % x2 < x1
                elseif final_offset(i+1,1) - final_offset(i,1) < 0
                    x_new = final_offset(i,1) - (seg_length*j);
                    y_new = final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end
                %-----------------------------------

            % Vertical Line --------------------
            elseif final_offset(i+1,1) - final_offset(i,1) == 0
                % y2 > y1
                if final_offset(i+1,2) - final_offset(i,2) > 0
                    x_new = final_offset(i,1);
                    y_new = final_offset(i,2) + (seg_length*j);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                    % y2 < y1
                elseif final_offset(i+1,2) - final_offset(i,2) < 0
                    x_new = final_offset(i,1);
                    y_new = final_offset(i,2) - (seg_length*j);
                    v_new2(count,1) = x_new;
```

```
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end
                %-----------------------------------

            % Diagonal Lines -------------------
            % x2 > x1
            elseif final_offset(i+1,1) - final_offset(i,1) > 0
                % y2 > y1
                if final_offset(i+1,2) - final_offset(i,2) > 0
                    m = (final_offset(i+1,2) - final_offset(i,2)) / (final_offset(i+1,1)
- final_offset(i,1));
                    x_new = final_offset(i,1) + ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif final_offset(i+1,2) - final_offset(i,2) < 0
                    m = (final_offset(i+1,2) - final_offset(i,2)) / (final_offset(i+1,1)
- final_offset(i,1));
                    x_new = final_offset(i,1) + ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end

            % x2 < x1
            elseif final_offset(i+1,1) - final_offset(i,1) < 0
                % y2 > y1
                if final_offset(i+1,2) - final_offset(i,2) > 0
                    m = (final_offset(i+1,2) - final_offset(i,2)) / (final_offset(i+1,1)
- final_offset(i,1));
                    x_new = final_offset(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif final_offset(i+1,2) - final_offset(i,2) < 0
                    m = (final_offset(i+1,2) - final_offset(i,2)) / (final_offset(i+1,1)
- final_offset(i,1));
                    x_new = final_offset(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end
            end
            %-----------------------------------
        end
    end
end

% Last coordinate to first coordinate
for i = final_offset_count
    dist = sqrt( ( final_offset(1,1)-final_offset(i,1) )^2 + ( final_offset(1,2)-
final_offset(i,2) )^2 );
    v_new2(count,1) = final_offset(i,1);
    v_new2(count,2) = final_offset(i,2);
    count = count + 1;

    if dist > offset_dist
        % Rounds up to determines number of segments to be made so that each segment is
        % less than "local offset dist" in length
        numb_seg = ceil( dist / local_offset_dist );
```

```
        % the length given to each new segment made
        seg_length = dist/numb_seg;
        % The number of new vertices to be added to the coordinate list.
        % The last vertex of the edge isn't MADE here, it is saved during
        % the next iteration of "i" as the vertex 0 for that edge.
        seg_created = numb_seg - 1;

        for j = 1:seg_created
            % Horizontal Line -------------------
            if final_offset(1,2) - final_offset(i,2) == 0
                % x2 > x1
                if final_offset(1,1) - final_offset(i,1) > 0
                    x_new = final_offset(i,1) + (seg_length*j);
                    y_new = final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % x2 < x1
                elseif final_offset(1,1) - final_offset(i,1) < 0
                    x_new = final_offset(i,1) - (seg_length*j);
                    y_new = final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end
                %-------------------------------------

            % Vertical Line --------------------
            elseif final_offset(1,1) - final_offset(i,1) == 0
                % y2 > y1
                if final_offset(1,2) - final_offset(i,2) > 0
                    x_new = final_offset(i,1);
                    y_new = final_offset(i,2) + (seg_length*j);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif final_offset(1,2) - final_offset(i,2) < 0
                    x_new = final_offset(i,1);
                    y_new = final_offset(i,2) - (seg_length*j);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end
                %-------------------------------------

            % Diagonal Lines -------------------
            % x2 > x1
            elseif final_offset(1,1) - final_offset(i,1) > 0
                % y2 > y1
                if final_offset(1,2) - final_offset(i,2) > 0
                    m = (final_offset(1,2) - final_offset(i,2)) / (final_offset(1,1) -
final_offset(i,1));
                    x_new = final_offset(i,1) + ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif final_offset(1,2) - final_offset(i,2) < 0
                    m = (final_offset(1,2) - final_offset(i,2)) / (final_offset(1,1) -
final_offset(i,1));
                    x_new = final_offset(i,1) + ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
```

```
                    count = count + 1;
                end

            % x2 < x1
            elseif final_offset(1,1) - final_offset(i,1) < 0
                % y2 > y1
                if final_offset(1+1,2) - final_offset(i,2) < 0
                    m = (final_offset(1,2) - final_offset(i,2)) / (final_offset(1,1) -
final_offset(i,1));
                    x_new = final_offset(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;

                % y2 < y1
                elseif final_offset(1,2) - final_offset(i,2) < 0
                    m = (final_offset(1,2) - final_offset(i,2)) / (final_offset(1,1) -
final_offset(i,1));
                    x_new = final_offset(i,1) - ((seg_length*j)/sqrt(1+m^2));
                    y_new = m*(x_new-final_offset(i,1)) + final_offset(i,2);
                    v_new2(count,1) = x_new;
                    v_new2(count,2) = y_new;
                    count = count + 1;
                end
            end
            %---------------------------------
        end
    end
end
v_new2;
% function end
end
```

Shaving.m

```
function[shave] = shaving(v_new2)

% This function will take the final coordinates of the offset polygon and
% will "soften" the corners by removing the points where two edges meet if
% they would not intersect with a hypothetical second round of offsetting.

global o_error
global CoordDirection

segment2_count = size(v_new2,1);

% Input coordinates placed into columns 2 (x) and 3 (y).
for i = 1:segment2_count
    data(i,2) = v_new2(i,1);
    data(i,3) = v_new2(i,2);
end

% Next to last row receives coordinates of the first coordinate point.
data(segment2_count+1,2) = v_new2(1,1);
data(segment2_count+1,3) = v_new2(1,2);
% Last row receives the coordinates of the second coordinate point
data(segment2_count+2,2) = v_new2(2,1);
data(segment2_count+2,3) = v_new2(2,2);

iteration = 1;
continue_loop = 1;
start1 = 1;
%---- Begin Iteration WHILE Loop ----
while continue_loop == 1
    iteration;
    %---- Selection of 3 coordinates for testing ----
    coord1 = 0;
    coord2 = 0;
    coord3 = 0;

    %---- Selection of first ----
    for i = start1:segment2_count
        if coord1 == 0
            first = i;
            % Stops the coord1 loop because a coordinate has been
            % chosen.
            coord1 = 1;
            start2 = first+1;
        end
    end
    %----------------------------

    %---- Selection of Second ----
    if coord1 == 1
        % If "first" is "1" then it can't use segment2_count+1 or +2 as "second" or
"third".
        if first == 1
            for i = start2:segment2_count
                if coord2 == 0
                    second = i;
                    % Stops the coord2 loop because a coordinate has been
                    % chosen.
                    coord2 = 1;
                    start3 = second+1;
                end
            end

        % If "first" is "2" then it can't use segment2_count+2 as "second" or "third".
        elseif first == 2
            for i = start2:segment2_count+1
```

```
                if coord2 == 0
                    second = i;
                    % Stops the coord2 loop because a coordinate has been
                    % chosen.
                    coord2 = 1;
                    start3 = second+1;
                end
            end

        % If "first" is anything except 1 or 2, then it is acceptable to use
segment2_count+1 and +2 as "second" or "third".
        else
            for i = start2:segment2_count+2
                if coord2 == 0
                    second = i;
                    % Stops the coord2 loop because a coordinate has been
                    % chosen.
                    coord2 = 1;
                    start3 = second+1;
                end
            end
        end
    end

    %---- Selection of Third ----
    if coord1 == 1
        if coord2 == 1
            % If "first" is "1" then it can't use segment2_count+1 or +2 as "second" or
"third".
            if first == 1
                for i = start3:segment2_count
                    if coord3 == 0
                        third = i;
                        coord3 = 1;
                    end
                end

            % If "first" is "2" then it can't use segment2_count+2 as "second" or
"third".
            elseif first == 2
                for i = start3:segment2_count+1
                    if coord3 == 0
                        third = i;
                        coord3 = 1;
                    end
                end

            % If "first" is anything except 1 or 2, then it is acceptable to use
segment2_count+1 and +2 as "second" or "third".
            else
                for i = start3:segment2_count+2
                    if coord3 == 0
                        third = i;
                        coord3 = 1;
                    end
                end
            end
        end
    end

    %---- Only calculates this section if three coordinates were chosen ----
    if coord1 == 1
        if coord2 == 1
            if coord3 == 1
                % If dd=0 then the lines are parallel. Don't run next steps
                % for parallel lines.
                one = [data(first,2) data(first,3)];
                two = [data(second,2) data(second,3)];
                three = [data(third,2) data(third,3)];
```

```
                dd = ( (data(third,3)-data(second,3))*(data(second,2)-data(first,2)) ) -
( (data(third,2)-data(second,2))*(data(second,3)-data(first,3)) );

        %---- Calculation of IEN for the chosen points ----
                % If the 2 lines are parallel
                if abs(dd) < o_error
                    disp('Lines in parallel');
                    iteration = iteration + 1;
                    start1 = start1 + 1;

                %if dd ~= 0 (NOT in parallel)
                elseif abs(dd) >= o_error
                    %---- First to Second ----
                    dx_12 = data(second,2) - data(first,2);
                    dy_12 = data(second,3) - data(first,3);
                    edgelength_12 = (dx_12*dx_12 + dy_12*dy_12)^(.5);

                    % Clockwise Coordinates
                    if CoordDirection == '1'
                        x_in_12 = dy_12/edgelength_12;
                        y_in_12 = -dx_12/edgelength_12;
                        data(first,4) = x_in_12;
                        data(first,5) = y_in_12;

                    % Counter-Clockwise Coordinates
                    elseif CoordDirection == '2'
                        x_in_12 = -dy_12/edgelength_12;
                        y_in_12 = dx_12/edgelength_12;
                        data(first,4) = x_in_12;
                        data(first,5) = y_in_12;
                    end

                    %---- Second to Third ----
                    dx_23 = data(third,2) - data(second,2);
                    dy_23 = data(third,3) - data(second,3);
                    edgelength_23 = (dx_23*dx_23 + dy_23*dy_23)^(.5);

                    % Clockwise Coordinates
                    if CoordDirection == '1'
                        x_in_23 = dy_23/edgelength_23;
                        y_in_23 = -dx_23/edgelength_23;
                        data(second,4) = x_in_23; %x of unit normal
                        data(second,5) = y_in_23;

                    % Counter-Clockwise Coordinates
                    elseif CoordDirection == '2'
                        x_in_23 = -dy_23/edgelength_23;
                        y_in_23 = dx_23/edgelength_23;
                        data(second,4) = x_in_23; %x of unit normal
                        data(second,5) = y_in_23;
                    end
                    %-----------------------------------------------------

        %---- Calculation of Mid point for the chosen points ----
                    %---- First to Second ----
                    % x-coord
                    data(first,6) = (data(first,2) + data(second,2))/2;
                    % y-coord
                    data(first,7) = (data(first,3) + data(second,3))/2;

                    %---- Second to Third ----
                    % x-coord
                    data(second,6) = (data(second,2) + data(third,2))/2;
                    % y-coord
                    data(second,7) = (data(second,3) + data(third,3))/2;
        %-------------------------------------------------------

        %---- Calculation of Intersection between 1-2's Midpoint/IEN with 2-3's
        %---- Midpoint/IEN ----
```

```
                    u = ((data(first,7)*data(second,4)) + (data(second,5)*data(second,6))
- (data(second,7)*data(second,4)) - (data(second,5)*data(first,6))) /
((data(first,4)*data(second,5))-(data(first,5)*data(second,4)));
                    v = (data(first,6)+(data(first,4)*u) -
data(second,6))/data(second,4);

                    % v is not calculated when the denominator is zero
                    if data(second,4) == 0
                        v = 1;
                    end

                    % If u and v are both positive then the rays will intersect
                    if u > 0 && v > 0
                        disp('intersection happens in ray directions')
                        start1 = start1 + 1;
                        iteration = iteration + 1;
                        data;

                    else %If u and/or v is less than 0 then there is no intersection.
                        % Shaves off the point of an angle so that
                        % the intersection of the offset edges will not split
                        % the final offset polygon into more than one polygon
                        data(second,1) = 100;
                        test_second = [data(second,2),data(second,3)];
                        disp('corner point shaved off')
                        start1 = start1 + 1;
                        iteration = iteration + 1;
                    end
                end
            end
        end
    end
    %-----------------------------------------------------

    %---- Stop While loop conditions ----
    %                Need coord2 or coord3 ?
    if coord1 == 0 || coord2 == 0 || coord3 ==0
        continue_loop = 0;
    end
    %-------------------------------------

% While end
end
%---- Coordinates 1 or 2 marked for deletion if correspinding extension coordinates are
marked ----
if data(segment2_count+1,1) == 100
    data(1,1) = 100;
end

if data(segment2_count+2,1) == 100
    data(2,1) = 100;
end
%-------------------------------------------------------------------------
%---- Simplifying Coordinate list ----
temp = v_new2;

for i = segment2_count:-1:1
    if data(i,1) == 100
        temp(i,:) = [];
    end
end
%-------------------------------------
shave = temp;
% Function end
end
```

Condense2.m

```
function [offset] = condense2(final)

% This subroutine will condense line segments in series that are parallel
% to each other into a single line segment
condense2_count = size(final,1);
global o_error

%---- Create temp matrix for coordinates to be marked for deletion ----
temp = zeros([condense2_count+2,3]);
for i = 1:condense2_count
    temp(i,2) = final(i,1);
    temp(i,3) = final(i,2);
end

% Next to last row receives coordinates of the first coordinate point.
temp(condense2_count+1,2) = final(1,1);
temp(condense2_count+1,3) = final(1,2);
% Last row receives the coordinates of the second coordinate point
temp(condense2_count+2,2) = final(2,1);
temp(condense2_count+2,3) = final(2,2);
%-----------------------------------------
%-----------------
iteration = 1;
continue_loop = 1;
start1 = 1;

%---- Begin Iteration WHILE Loop ----
while continue_loop == 1
    iteration;
    %---- Selection of 3 coordinates for testing ----
    coord1 = 0;
    coord2 = 0;
    coord3 = 0;

    %---- Selection of first ----
    for i = start1:condense2_count
        if coord1 == 0
            if temp(i,1) == 0
                first = i;
                % Stops the coord1 loop because a coordinate has been
                % chosen.
                coord1 = 1;
                start2 = first+1;
            end
        end
    end
    %----------------------------

    %---- Selection of Second ----
    if coord1 == 1
        % If "first" is "1" then it can't use simple_count+1 or +2 as "second" or
"third".
        if first == 1
            for i = start2:condense2_count
                if coord2 == 0
                    if temp(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end
```

```
        % If "first" is "2" then it can't use simple_count+2 as "second" or "third".
        elseif first == 2
            for i = start2:condense2_count+1
                if coord2 == 0
                    if temp(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end

        % If "first" is anything except 1 or 2, then it is acceptable to use
simple_count+1 and +2 as "second" or "third".
        else
            for i = start2:condense2_count+2
                if coord2 == 0
                    if temp(i,1) == 0
                        second = i;
                        % Stops the coord2 loop because a coordinate has been
                        % chosen.
                        coord2 = 1;
                        start3 = second+1;
                    end
                end
            end
        end
    end

    %---- Selection of Third ----
    if coord1 == 1
        if coord2 == 1
            % If "first" is "1" then it can't use simple_count+1 or +2 as "second" or
"third".
            if first == 1
                for i = start3:condense2_count
                    if coord3 == 0
                        if temp(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end

            % If "first" is "2" then it can't use simple_count+2 as "second" or "third".
            elseif first == 2
                for i = start3:condense2_count+1
                    if coord3 == 0
                        if temp(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end

            % If "first" is anything except 1 or 2, then it is acceptable to use
simple_count+1 and +2 as "second" or "third".
            else
                for i = start3:condense2_count+2
                    if coord3 == 0
                        if temp(i,1) == 0
                            third = i;
                            coord3 = 1;
                        end
                    end
                end
            end
```

```
        end
    end

    %---- Only calculates this section if three coordinates were chosen ----
    if coord1 == 1
        if coord2 == 1
            if coord3 == 1
                dd = ( (temp(third,3)-temp(second,3))*(temp(second,2)-temp(first,2)) ) -
( (temp(third,2)-temp(second,2))*(temp(second,3)-temp(first,3)) );
                % If dd=0 then the lines are parallel.
                % parallel lines not condensed here cause problems later.

                % Error allowance
                if abs(dd) < o_error
                    temp(second,1) = 100;
                    iteration = iteration + 1;

                % Not in parallel
                % Error allowance
                elseif abs(dd) >= o_error
                    start1 = start1 + 1;
                    iteration = iteration + 1;
                end
            end
        end
    end

    %---- Coordinates 1 or 2 marked for deletion if correspinding extension coordinates
are marked ----
    if temp(condense2_count+1,1) == 100
        temp(1,1) = 100;
    end

    if temp(condense2_count+2,1) == 100
        temp(2,1) = 100;
    end
    %-------------------------------------------------------------------------

    %---- Stop While loop conditions ----
    %                Need coord2 or coord3 ?
    if coord1 == 0 || coord2 == 0 || coord3 ==0
        continue_loop = 0;
    end
    %-------------------------------------

% While loop end
end

%---- Simplifying Coordinate list ----
temp2 = final;

for i = condense2_count:-1:1
    if temp(i,1) == 100
        temp2(i,:) = [];

    end
end
%-------------------------------------
offset = temp2;
% Function end
end
```

References

Agioutantis, Z. and Karmis, M. (2013). "Addressing the effect of sloping terrain on ground movements due to underground mining." In: Craynon, J. R., Ed. *Proceedings of the Symposium on Environmental Considerations in Energy Production*. Blacksburg, VA: Appalachian Research Initiative for Environmental Sciences, pp. 308–318.

Agioutantis, Z. and Karmis, M. (2016). "Quick reference guide and working examples." *Surface Deformation Prediction System for Windows*. Version 6.2D. Blacksburg, VA: Virginia Polytechnic Institute and State University, 330p.

Bell, F.G. *Engineering in Rock Masses*. Elsevier, 2013, 592p.

Chen, X. and McMains, S. (2005). "Polygon offsetting by computing winding numbers." In: *Proceedings of International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. September 24-28, 2005, Long Beach, California, USA.

Karmis, M. and Agioutantis, Z. (2015). "A methodology to assess the potential impacts of longwall mining on streams in the Appalachian Basin." In: *Proceedings of the 2nd Conference on Environmental Considerations in Energy Production*. Golden, CO: Society for Mining, Metallurgy, and Exploration.

Karmis, M., Agioutantis, Z., and Andrews, K. (2008). "Enhancing mine subsidence prediction and control methodologies." In: *Proceedings of the 27th International Conference on Ground Control in Mining*. Morgantown, WV: West Virginia University, pp. 131–136.

Knothe, S., (1957). "Observations of surface movements under influence of mining and their theoretical interpretation." *Proceedings of the European Congress on Ground Movement*. Leeds, UK: University of Leeds, pp. 210–218.

Kratzsch, H. *Mining Subsidence Engineering*. New York: Berlin Heidelberg, 1983, 543p.

Newman, C., Agioutantis, Z., Bode Jimenez Leon, G., and Karmis M. (2016). "Evaluation of potential impacts to stream and ground water due to underground coal mining." In: *Proceedings of the 35th International Conference on Ground Control in Mining*. Morgantown, WV: West Virginia University, pp. 8.

Newman, D., Agioutantis, Z., and Karmis, M. (2001). “SDPS for Windows: An integrated approach to ground deformation prediction.” In: *Proceedings of the 20th International Conference on Ground Control in Mining*. Morgantown, WV: West Virginia University, pp. 6.

VPI&SU. (1987). *Prediction of Ground Movements Due to Underground Mining in the Eastern United States Coalfields*. Final Report Contract No. J5140137. Volumes I and II. Washington DC: U.S. Department of the Interior, Office of Surface Mining, Reclamation, and Enforcement. Vol I, 205p. Vol II, 112p.

Vita

Joshua Hescock

Education

Master of Engineering, Mining Engineering, University of Kentucky, Lexington Kentcuky.

Bachelor of Science, Mining Engineering, University of Kentucky, Lexington, KY. May 2015.

Publications

Hescock, J., Agioutantis, Z. (2017). "SDPS Update: Easy calculation of the edge effect offset for complex extraction panels." In: *Proceedings of the 36th International Conference on Ground Control in Mining*. Morgantown, WV: West Virginia University, pp. 353-358.