U.S. DEPARTMENT OF ENERGY

PNNL-21893

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Methods for Data-based Delineation of Spatial Regions

JE Wilson

October 2012

Pacific Northwest
NATIONAL LABORATORY

Proudly Operated by Battelle *Since 1965*

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights**. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service,
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

This document was printed on recycled paper.
(9/2003)

PNNL-21893

Methods for Data-based Delineation of Spatial Regions

JE Wilson

October 2012

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99352

Summary

In data analysis, it is often useful to delineate or segregate areas of interest from the general population of data in order to concentrate further analysis efforts on smaller areas. Three methods are presented here for automatically generating polygons around spatial data of interest. Each method addresses a distinct data type. These methods were developed for and implemented in the sample planning tool called Visual Sample Plan.

Method A is used to delineate areas of elevated values in a rectangular grid of data (raster). The data used for this method are spatially related. Although Visual Sample Plan uses data from a kriging process for this method, it will work for any type of data that is spatially coherent and appears on a regular grid.

Method B is used to surround areas of interest characterized by individual data points that are congregated within a certain distance of each other. Areas where data are "clumped" together spatially will be delineated.

Method C is used to recreate the original boundary in a raster of data that separated data values from non-values. This is useful when a rectangular raster of data contains non-values (missing data) that indicate they were outside of some original boundary. If the original boundary is not delivered with the raster, this method will approximate the original boundary.

Appendices contain C++ code for each of the methods. The appendices may not contain all the supporting functions called by the core functions, however, the most important and non-obvious supporting functions are presented.

Acknowledgments

I would like to thank Brent A. Pulsipher and John E. Hathaway for their support and help on the project.

Contents

Figures

1.0 Method A

This method is used to delineate areas of values above (or possibly below) a threshold level in a spatially related regular grid. Figure 1 depicts such a grid of values, where each square in the grid is colored according to its Kriged value. Kriging is a method of estimating a value on a regular grid by interpolating between sparsely located measured values.

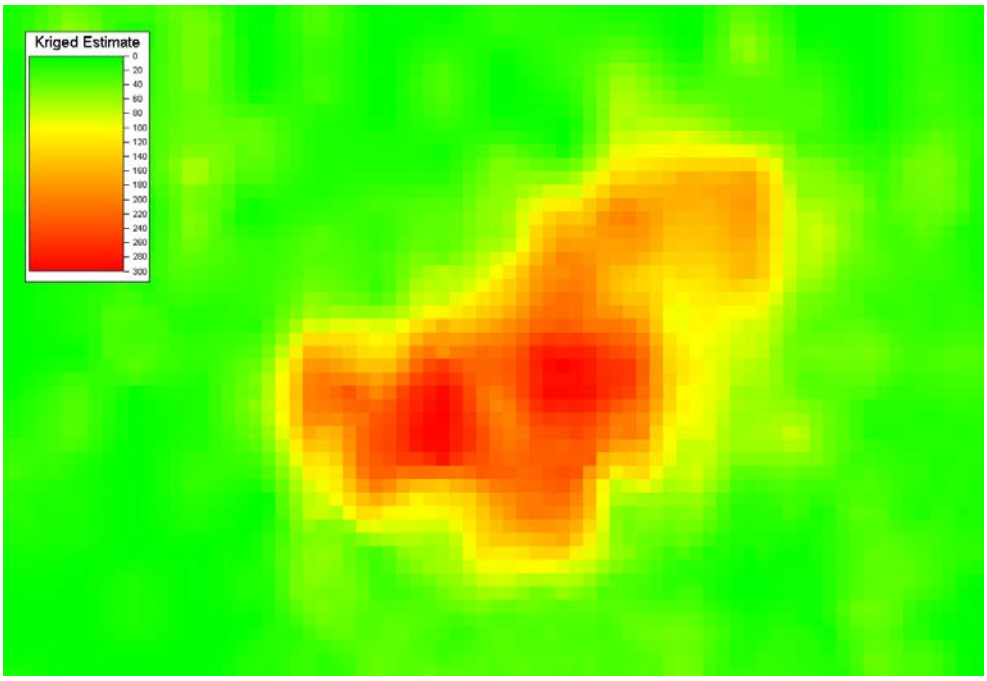

Figure 1. Kriged Values

1. The process begins by selecting those cells from the grid above a user-specified threshold. For this example, those cells that are greater than or equal to 80 units are selected. Figure 2 below illustrates the selected cells in black.

Figure 2. Cells Greater than Specified Threshold

2. At this point, all the selected cells are independent, having no spatial relationship to each other. The following algorithm is performed to group these individual cells into "clusters".

   a. The cells are sorted by their Y (vertical) coordinate and then by their X (horizontal) coordinate to make the clustering algorithm more efficient.

   b. The first cell is removed from the sorted list. (For efficiency, the cell is not actually removed but is marked as "used".) The first cell is also placed on a stack (first-in, last-out list) so that further investigation can be conducted at the location. All unused cells that are part of the same "cluster" as the first cell are added to the cluster as follows:

      i. If there is a cell in the stack, it is removed from the stack and becomes the "current cell". If there are no cells in the stack, then the cluster is complete and the algorithm ends. Continue at step A.2.c.

      ii. The location above the "current cell" is checked. If the cell for that location is in the unused list, that cell is removed from the list and added to the current cluster. That cell is also placed on the stack.

      iii. The process in step A.2.ii is repeated for locations to the right, below, left, above and right, below and right, below and left, above and left of the "current cell".

      iv. Steps A.2.b.i through A.2.b.iii are repeated until there are no more locations in the stack.

   c. If there are any unused cells left in the list, then step A.2.b is repeated creating another cluster. If there are no unused cells left in the list, then the creation of clusters is complete.

3. At this point, all the cells are contained in clusters. The next step is to create a polygon that represents the outer edge of each cluster. The following algorithm accomplishes this.

   a. A regular grid is established for the cluster. Each cell in the grid is set on or off, depending on whether a cell in the cluster exists at that location. Figure 3 below depicts the regular grid for a cluster.

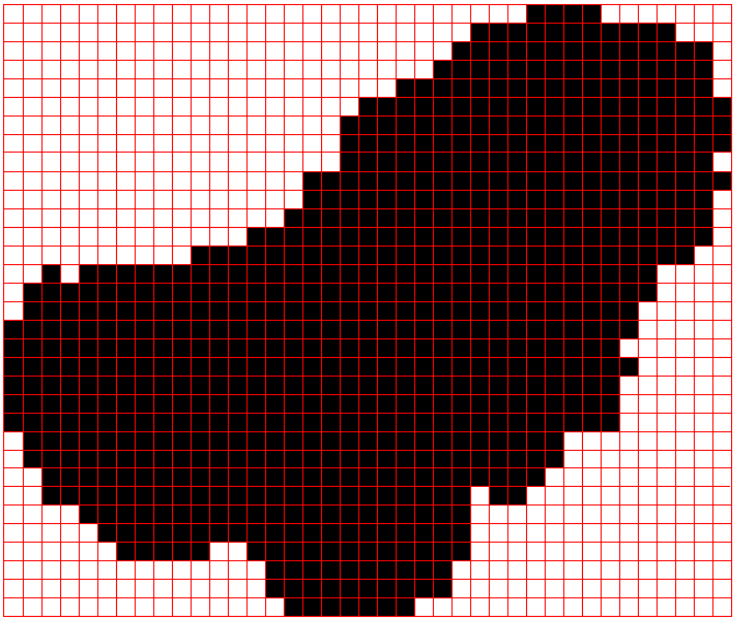

Figure 3. Regular Grid for Cluster. Black cells are part of cluster, white cells are not.

b. A check is done to find any locations where part of a cluster is joined together only by a corner. If such a location is found it is filled in as illustrated in Figure 4 below. This is done to avoid inner loops or missing sections in the next part of the algorithm. After a section is added, the entire grid is search again to ensure that no more occurrences were added by the action.

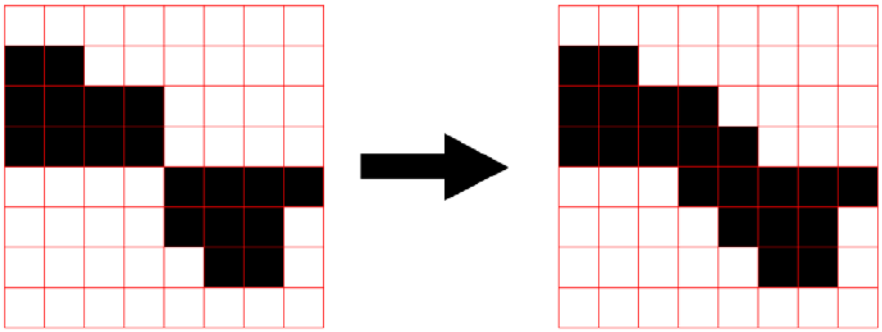

Figure 4. Filling in Section Joined by Corner

c. Next, everywhere that a cluster cell is next to a blank cell, that edge line segment is added to a segment list. Figure 5 depicts all the edges in the segment list where cluster cells border blank cells.

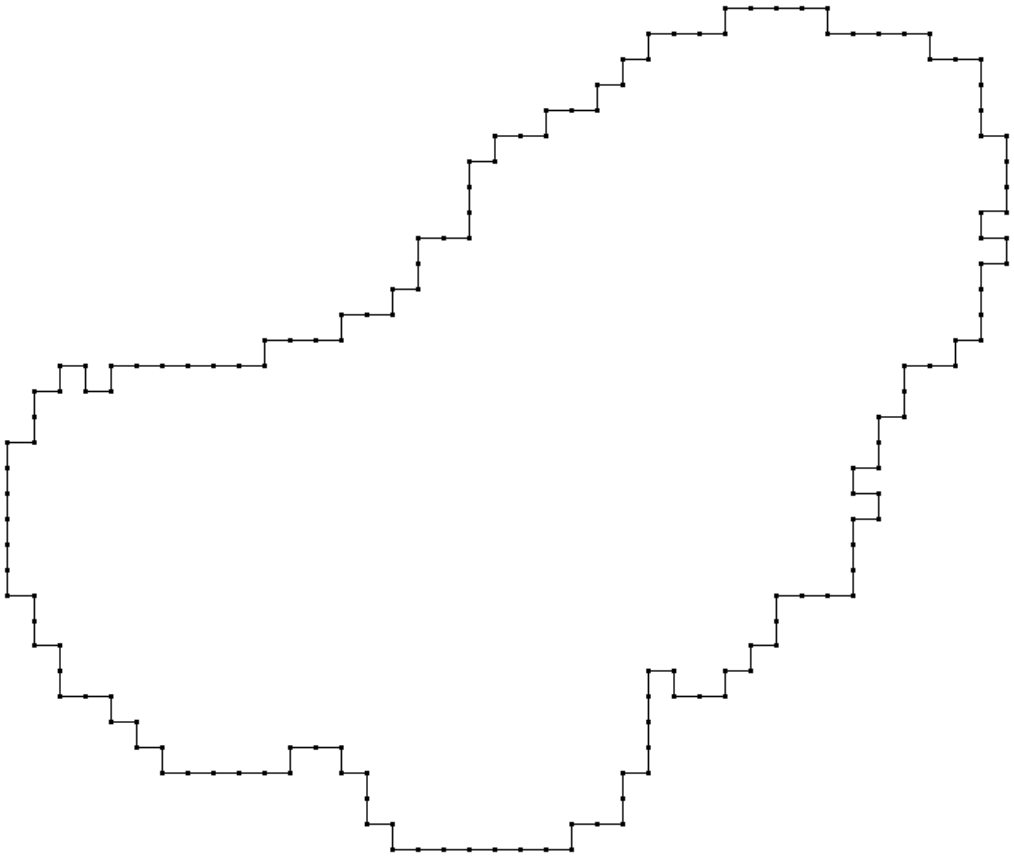

Figure 5. Edges Where Cluster Cells Adjoin Non-cluster Cells

d. Segments in the list are joined end-to-end in a polygon until the end meets the beginning, creating a closed polygon. This method of creating a polygon automatically eliminates interior holes, because the first segment chosen is on the outer polygon and the algorithm ends when the outer polygon is closed.

e. Finally, collinear points are removed.

4. Now the boundary of the elevated area is defined by a polygon. However, the polygon is very jagged. The following algorithm simplifies the polygon and makes it more aesthetic.

   a. Each angle created by 3 points is checked. If the angle is greater than 90 degrees, the angle is ignored and next angle is checked.

   b. The line segment from the first point to the third point is checked. If the segment lies inside the polygon, the angle is ignored and the next angle is checked (step A.4.a).

   c. The triangle formed by the 3 points is checked to see if it contains any other point in the polygon. If it does, the angle is ignored and the next angle is checked (step A.4.a). Removal of such an angle could result in a polygon that crosses over itself.

   d. If the angle passes the tests listed above, the middle point is removed, straightening out the edge. Figure 6 depicts the polygon after the first pass of the algorithm. The red segments represent where edges have been straightened by removing the middle point of an angle.

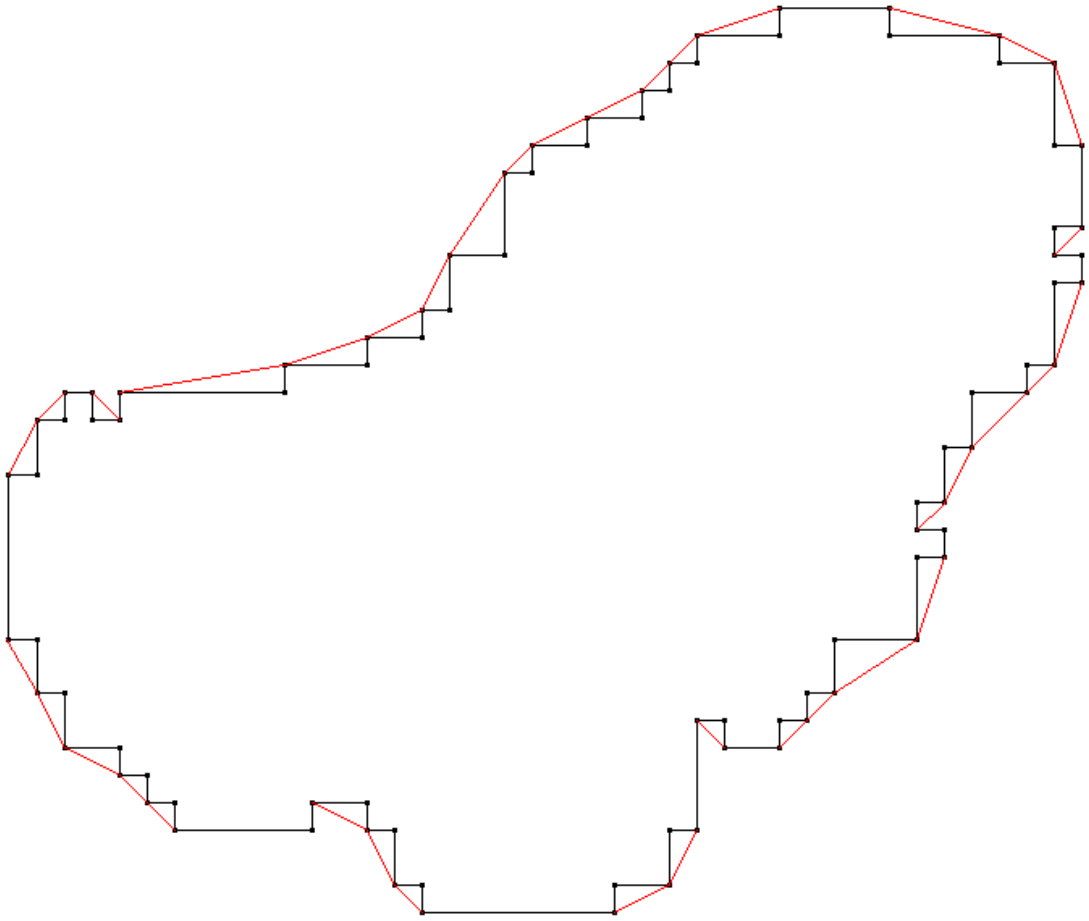

Figure 6. Straightening and Simplifying the Polygon

e. After all the angles in the polygon have been checked, the algorithm is repeated (steps A.4.a through A.4.d) until no more points are removed.

f. Finally, collinear points (that might have been created by the algorithm) are removed. Figure 7 represents the final polygon created by the algorithm.

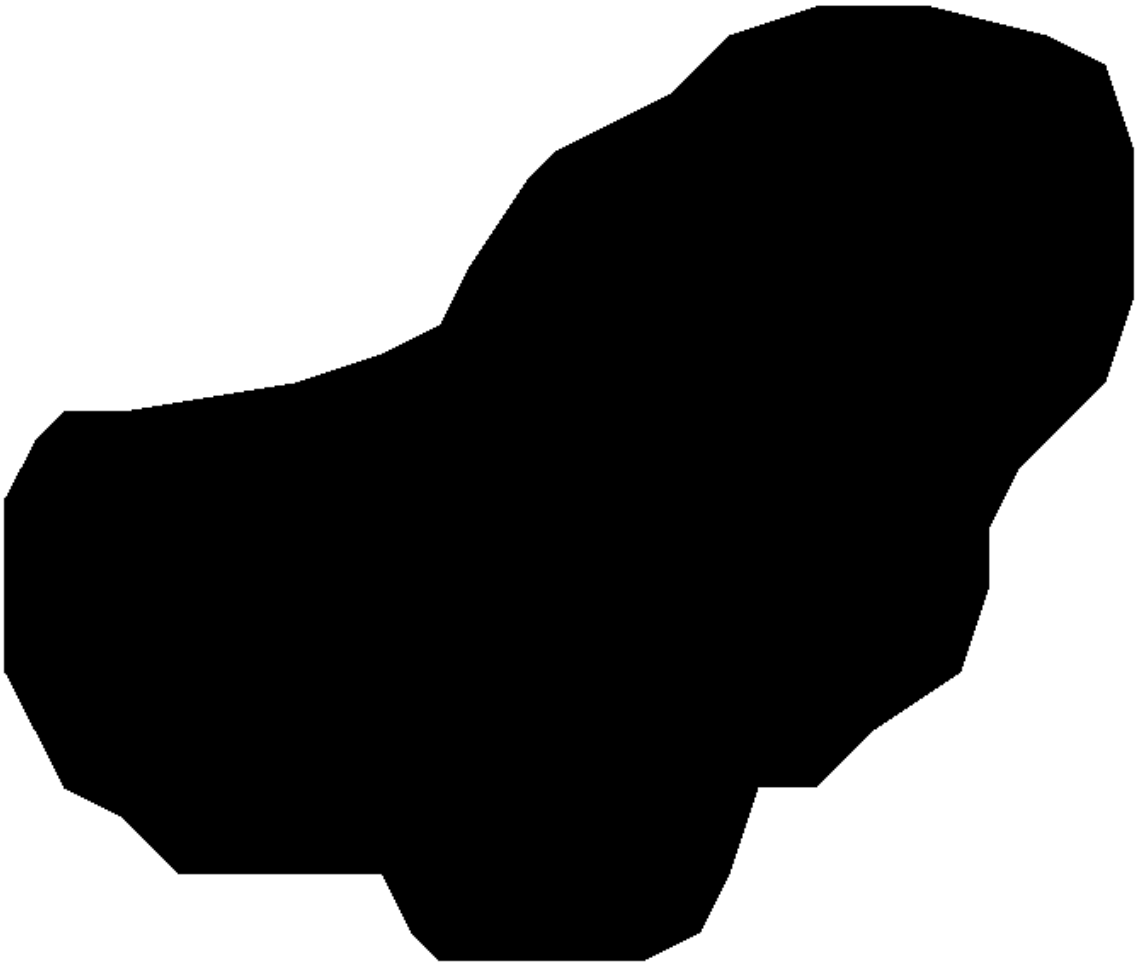

Figure 7. The Final Polygon Representing the Area of Elevated Density

1.1 Method A Conclusions

This method is guaranteed to contain all the values above the threshold. The process of removing interior acute and right angles will contain areas below the threshold. However, this area will be no larger (and usually much smaller) than the convex hull of the same polygon.

2.0 Method B

This method is used to surround areas of interest characterized by individual data points that are congregated within a certain distance of each other. The purpose or meaning of the data points is not considered. This method assumes that irrelevant data points have been removed prior to the method. Areas where data are "clumped" together within a user-specified distance will be delineated.

Figure 8 below depicts data points, or markers, that have been placed on a map in Visual Sample Plan.

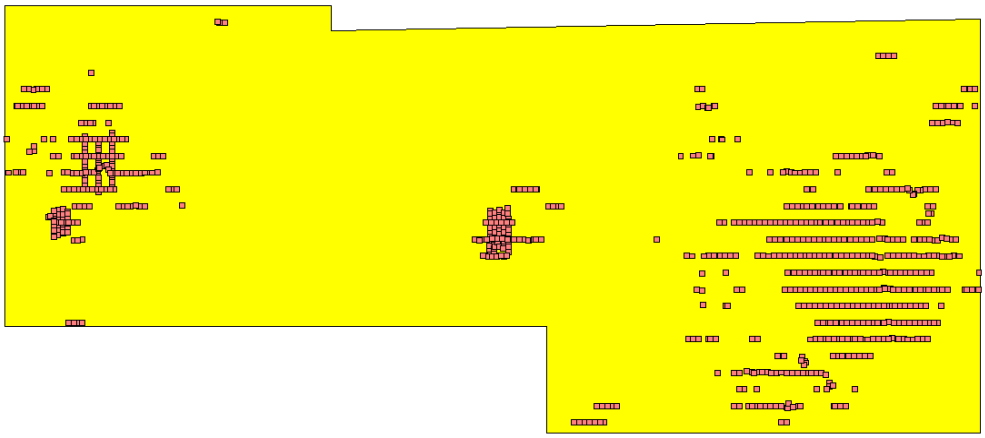

Figure 8. Markers Placed on a Map

1. The process begins by creating a square of a user-specified size around each data point. The squares are placed into a list and are sorted by their Y (vertical) coordinate then by their X (horizontal) coordinate to improve the efficiency of subsequent steps in the process. Figure 9 below visualizes the arrangement of these squares.

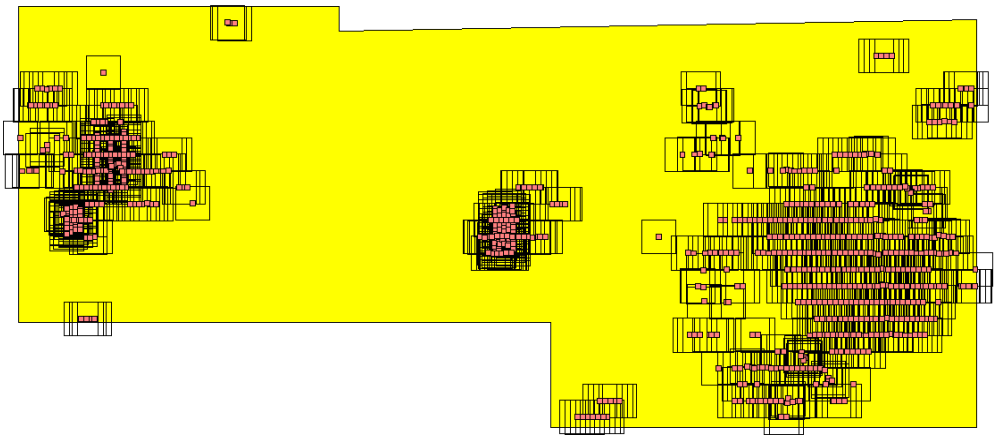

Figure 9. Squares Around Markers

2. Next, the squares are grouped together into polygons. Squares touching each other become part of the same polygon. This is accomplished by the following algorithm:

a. The first square is removed from the list and becomes the “current polygon”. (For efficiency, the square is marked as being used rather than actually being removed from the list.) This square is also placed on a stack (first-in, last-out list) so that further investigation can be conducted at the location.

   i. If there is a square in the stack, it is removed from the stack and becomes the “current square”. If there are no squares in the stack, then the polygon is complete and the algorithm ends. Continue at step B.2.b.

   ii. This list of unused squares is searched in both directions from the “current square” to see if any unused squares overlap the “current polygon”. (This searching is made efficient by the sorted list and proximity checks.) Each unused square that does overlap is removed from the list and added to the “current polygon” by a polygon union. (The polygon union is not discussed here.) These squares are also placed on the stack.

   iii. Steps B.2.a.i through B.2.a.ii are repeated until there are no more squares on the stack.

b. The “current polygon” is saved. If there are any unused squares left in the list, step B.2.a is repeated to create another polygon. Figure 10 represents the resulting polygons.

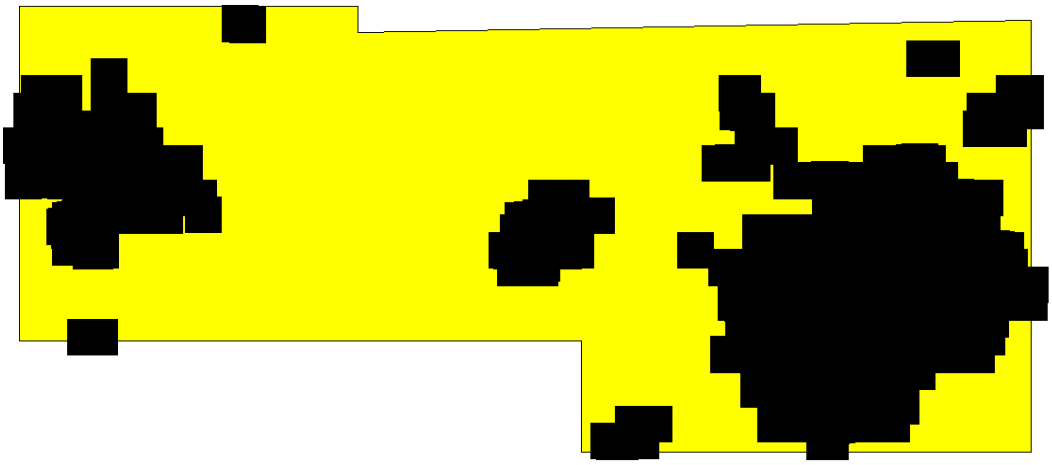

Figure 10. Squares Joined Together into Polygons

3. Now the boundary of each elevated area is defined by a polygon. However, the polygons are very jagged. The following algorithm simplifies the polygons and makes them more aesthetic.

   a. Each angle created by 3 points is checked. If the angle is greater than 110 degrees, the angle is ignored and next angle is checked. This threshold angle could also be a user input.

   b. The line segment from the first point to the third point is checked. If the segment lies inside the polygon, the angle is ignored and the next angle is checked (step a).

   c. The triangle formed by the 3 points is checked to see if it contains any other point in the polygon. If it does, the angle is ignored and the next angle is checked (step B.3.a). Removal of such an angle could result in a polygon that crosses over itself.

   d. If the angle passes the tests listed above, the middle point is removed, straightening out the edge. Figure 11 depicts a polygon after the first pass of the algorithm. The red segments represent where edges have been straightened by removing the middle point of an angle.

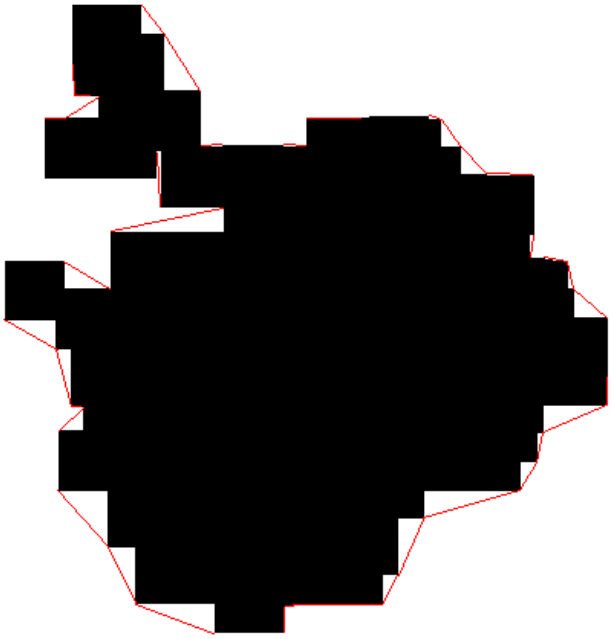

Figure 11. Simplifying the Polygon

e. After all the angles in the polygon have been checked, the algorithm is repeated (steps B.3.a through B.3.d) until no more points are removed.

f. Finally, collinear points (that might have been created by the algorithm) are removed. Figure 12 represents the final polygons created by the algorithm.

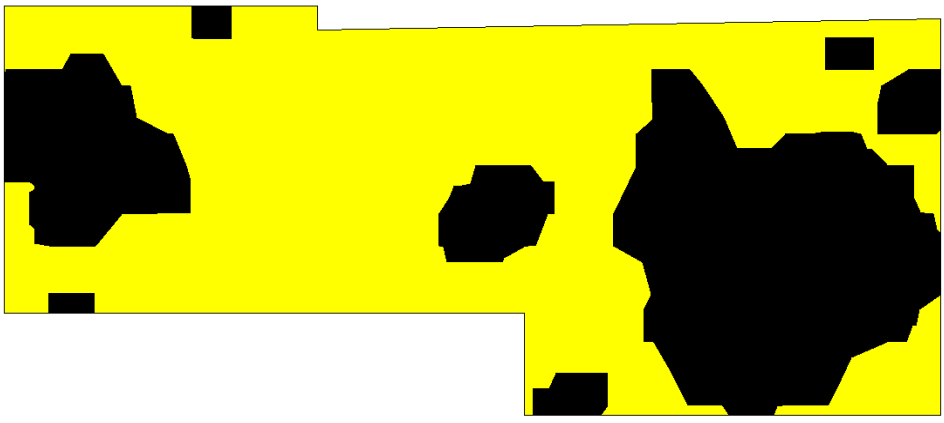

Figure 12. The Final Polygon Representing the Areas of Interest

2.1 Method B Conclusions

The process of removing interior acute and right angles results in an area that might be larger than is strictly necessary to contain all the points. However, this area will be no larger (and usually much smaller) than the convex hull of the same polygon. It might be useful to try surrounding each data point with a shape other than a square. A hexagon might yield aesthetically pleasing results. The angle used in step B.3.a could be varied. The current setting is 110 degrees to remove interior right angles.

3.0 Method C

The third method (Method C) is used to recreate the original boundary in a raster of data that separated data values from non-values. This is useful when a rectangular raster of data contains non-values (missing data) that indicate they were outside of some original boundary.

Figure 13 depicts such a pattern of data values. The dots represent a regular grid of data locations. Many dots are missing at the edges of the pattern indicating locations outside of the original area of interest. Algorithm C will reconstruct the approximate boundary of the original area.

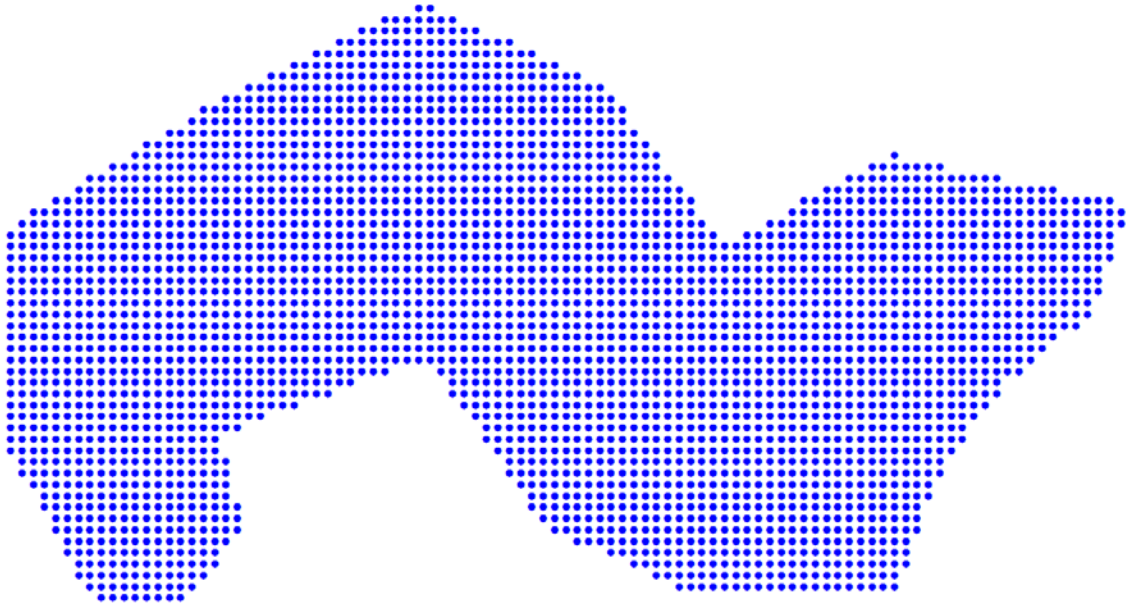

Figure 13. Pattern of Data Inside the Area of Interest

1. The process begins by finding all the non-blank points that are adjacent to blank points. Following from natural order of mapping coordinates, the points are ordered bottom to top then left to right. Because the data are placed on a regular grid, the points in this step are actually row and column indices, rather than spatial coordinates. The points will be rectified to actual spatial coordinates in a later step. Figure 14 shows the pattern of resulting points. If a point stands alone on the bottom row it is removed (because such a point will interfere with the next step).

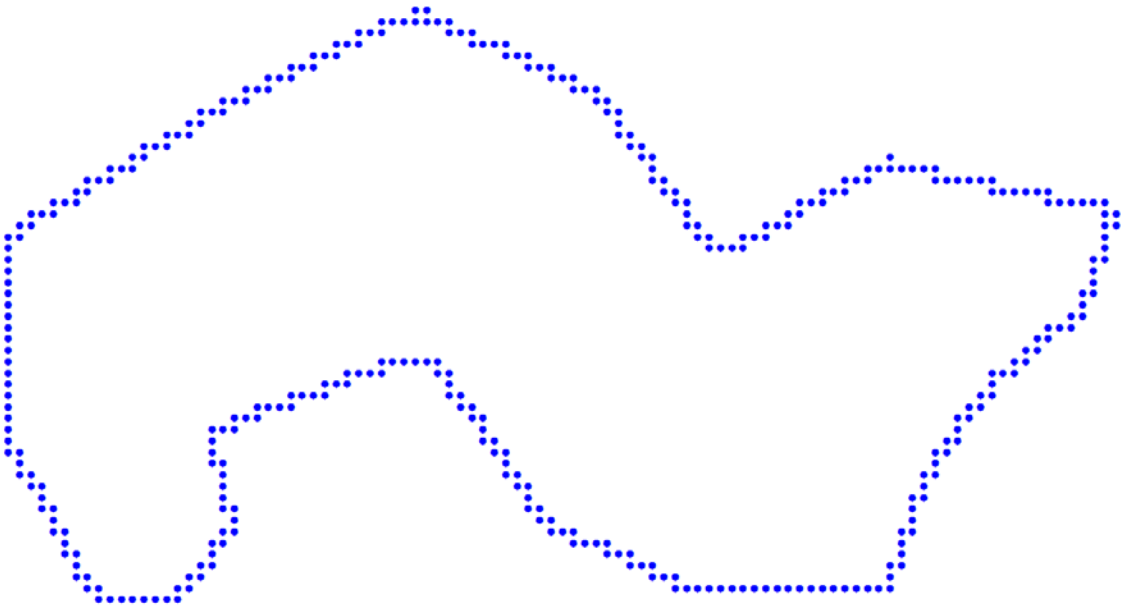

Figure 14. Pattern of Non-blank Points that are Adjacent to Blank Points

2. Next, these points are joined together to make a polygon. The constraints on the polygon are: 1) points are connected either vertically or horizontally but not diagonally, 2) consecutive points are in adjacent columns or rows, 3) if a point stands alone on a row or column then it must be duplicated in the final polygon (see Figure 15), and 4) the polygon will naturally be ordered counter-clockwise.

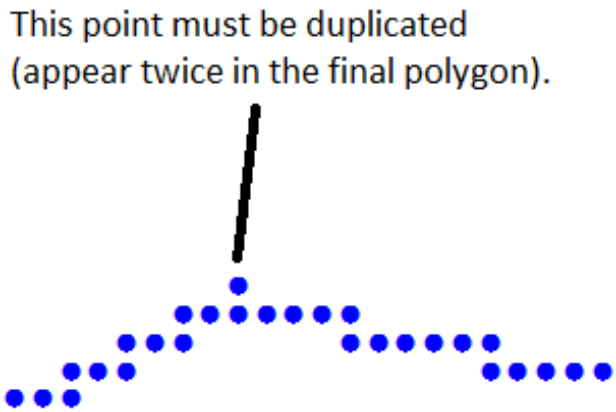

Figure 15. Point Standing Alone in Row that Must be Duplicated

   a. The first point (bottom row and left-most column of data) is selected as the current point.

   b. The current point is added to the polygon and removed from the available list of points.

   c. The taxi distance is computed from the current point to each available point. The taxi distance is the absolute column difference plus the absolute row distance. The minimum taxi distance is computed.

d. If the minimum taxi distance is greater than one then the polygon must be complete and the current process ends. Continue with step C3. Any remaining points constitute a contained hole. They may be discarded or processed as desired. Contained holes will be discarded for our purposes.

e. If more than one point shares the minimum taxi distance of one, then the point with a single connection remaining is chosen as the minimum point (see Figure 16). If more than one point has a single connection remaining, then the right-most point is chosen. If a point has no available connections, then this it is the special case where a point doubles back on itself. In such a case, the point will be added to the polygon and the previous polygon point will be added back to the list of available points allowing the polygon to double back on itself and continue from there.

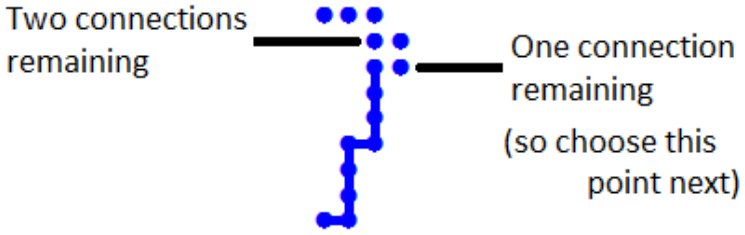

Figure 16. Choose Point with Fewest Remaining Connections

f. The minimum point becomes the current point and processing continues at step C.2.b.

3. Rectify the list of connected index points into a regular polygon that constitutes the inner boundary.

   a. Each row column index point is rectified into a spatial point:

      i. X = minimum x + column * column spacing

      ii. Y = minimum y + row * row spacing

   b. Co-linear points are removed

4. Next, all blank points that are adjacent to non-blank points are collected in a list. This is similar to step C.1. Special care must be taken because the points may be outside the boundaries of the grid.

5. Step C.2 and C.3 will be performed on this list of index points to produce a polygon than constitutes the outer boundary. The final boundary will be constrained between the inner and outer boundaries (although the original boundary may not have been completely constrained between them). Figure 17 shows the inner and outer boundaries. For the rest of the algorithm to work properly, both boundary polygons must have the same number of vertices. This is the reason why points must be duplicated when the boundary doubles back on itself (see Figure 18).

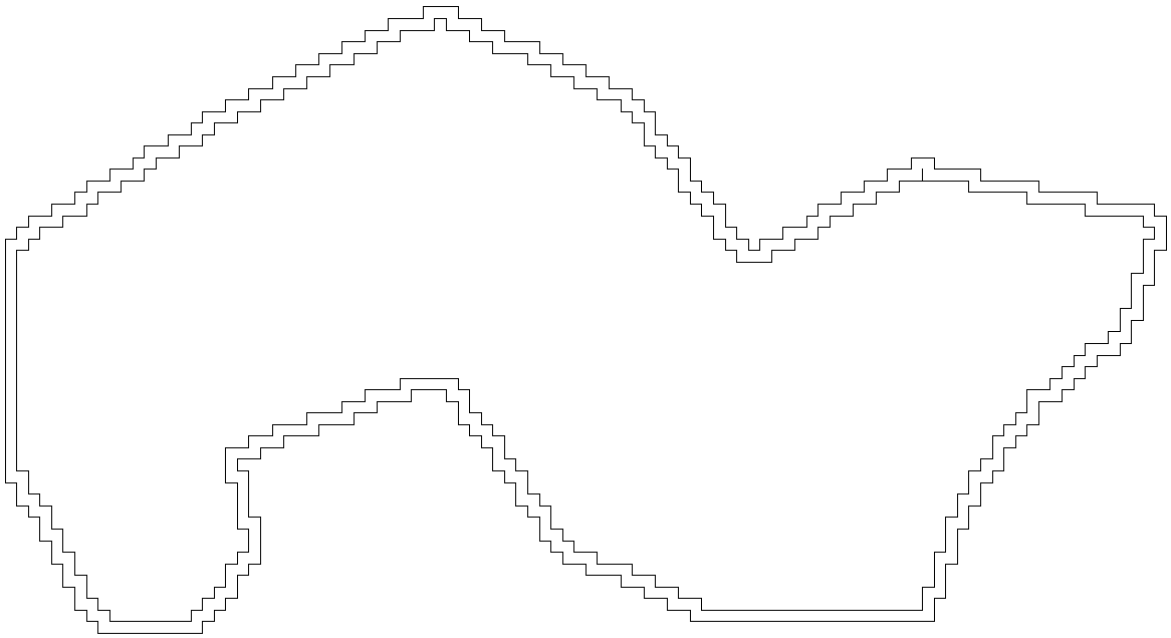

Figure 17. Inner and Outer Boundaries

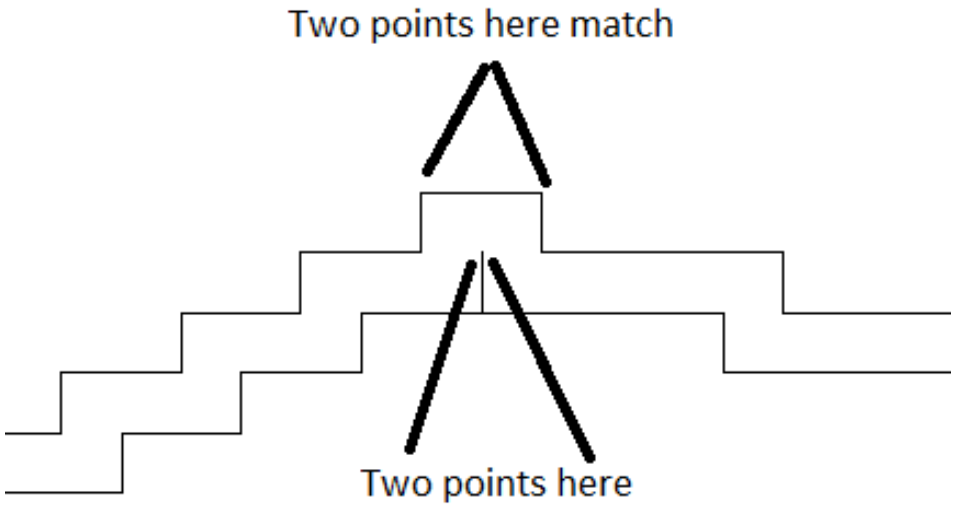

Figure 18. Points Must be Duplicated to Match Inner and Outer Boundaries

6. Next, the polygons are examined and reversal points are identified and added to two lists: an inner and an outer list. A reversal is where two consecutive points have 90 degree turns in the same direction. An outer reversal has two turns towards the center of the polygon and an inner reversal has two turns away from the center of the polygon. Figure 19 shows the outer reversals marked with red dots and inner reversals marked with green dots. The reversal lists contain the point (vertex) number of the first of the two turns. The red arrow points to a reversal point so defined.

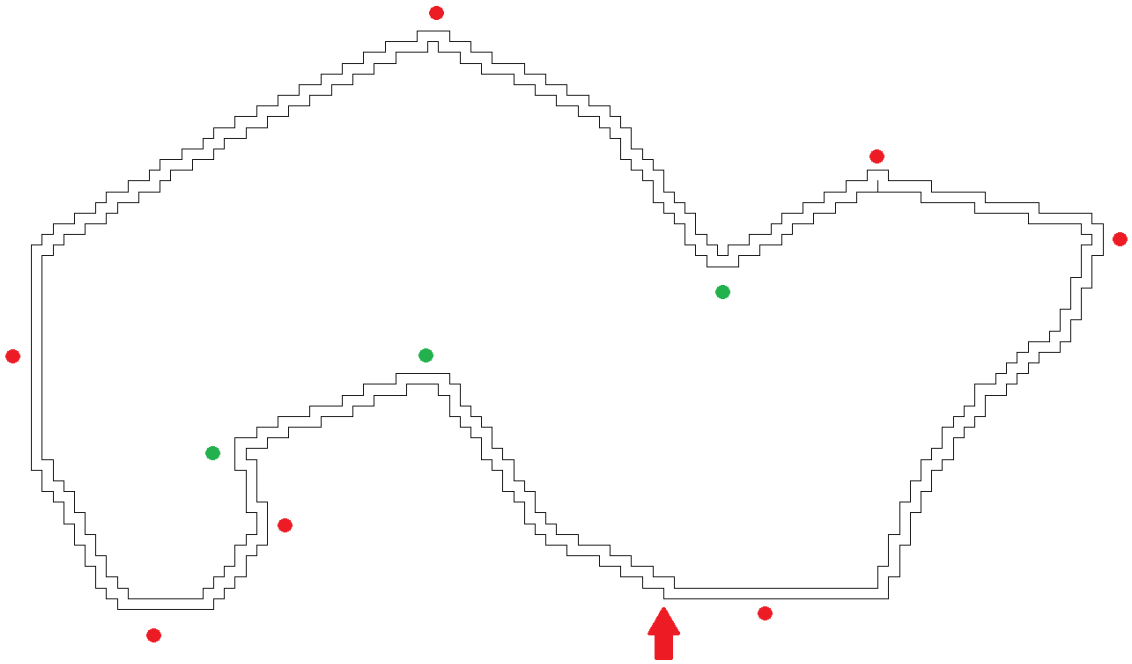

Figure 19. Outer and Inner Reversals Marked with Red and Green Dots, Respectively

7. The next step of the algorithm, in simple terms, chooses a beginning point and adds it to the final boundary polygon and then tries to extend a line from the beginning point down the corridor formed by the inner and outer boundaries until the line can be extended no further without hitting a boundary. This end is added to the final boundary and becomes a new beginning point and the process is repeated. The details of the algorithm are as follows:

   a. The "beginning" point is chosen as the midpoint of the first two points on the outer boundary (see Figure 20). This point is stored to the boundary.

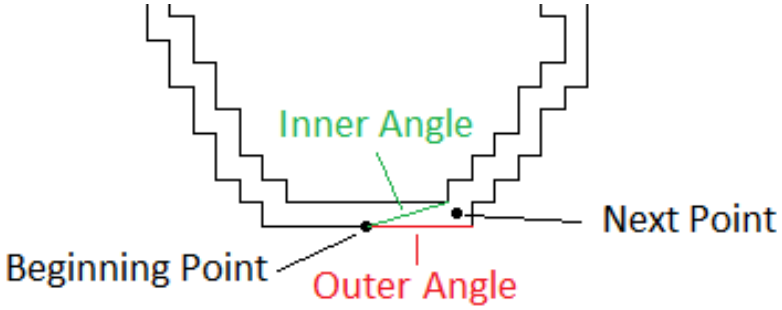

Figure 20. Location of Beginning and Next Points with Inner and Outer Angles

   b. An inner index and an outer index are set to the 2nd point of the inner boundary and outer boundary, respectively.

   c. The angle from the beginning point to the point at the inner index is computed and stored as the inner angle. The angle from the beginning point to the point at the outer index is computed and stored as the outer angle (see Figure 20).

   d. The midpoint of the points at the inner index and outer index is computed and stored as the "next" point (see Figure 20).

e. Check to see if the inner index is at the end of the inner polygon. If so, close the boundary and the algorithm is complete. See figures 22 and 23.

f. Check to see if the inner angle is less than or equal to the outer angle. If so, this means that the line has progressed as far as possible down the corridor between the inner and outer boundaries. Therefore:

   i. add the "next" point to the boundary

   ii. set the "beginning" point equal to the "next" point

   iii. move the inner and outer index one past the "next" point

   iv. check to see if the inner index is at the next inner reversal. If so, navigate out of the reversal with these steps:

      1. set the "next" point equal to the midpoint of the points at the inner and outer index

      2. add the "next" point to the boundary

      3. increment the inner and outer indices

      4. set the "next" point equal to the midpoint of the points at the inner and outer index

      5. add the "next" point to the boundary

      6. increment the inner and outer indices

   v. set the "next" point equal to the midpoint of the points at the inner and outer index

   vi. set the inner and outer angles from the new "beginning" point to the points at the inner and outer indices respectively

g. Increment the inner index.

h. See if the inner index is at the next inner reversal. If so:

   i. compute the mid angle as the average of the inner and outer angles (adjust outer angle if it not pointing to the outer boundary opposite the inner reversal)

   ii. if the mid angle intersects the centerline of the reversal (see Figure 21) then set the "next" point to the intersection

   iii. otherwise, if the mid angle intersects the edge of the reversal (see Figure 21) then set the "next" point to the intersection

   iv. add the "next" point to the boundary

   v. set the "beginning" point equal to the "next" point

   vi. set the inner and outer indices to the point past the reversal

   vii. set the "next" point equal to the midpoint of the points at the inner and outer index

   viii. set the inner and outer angle from the new "beginning" point to the points at the inner and outer indices respectively.

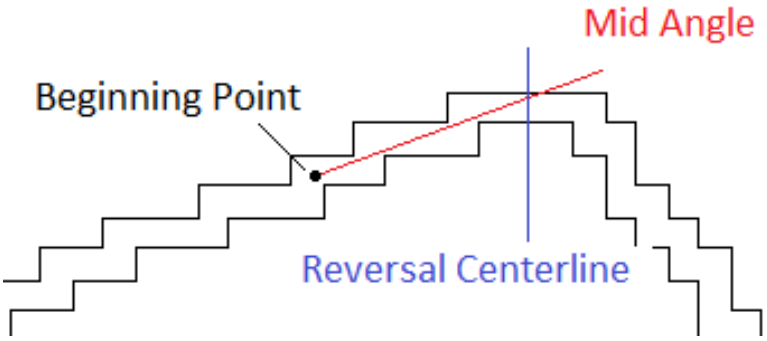

Figure 21. Location of Reversal Centerline with Respect to Mid Angle and Beginning Point

i. Calculate the angle from the "beginning" point to the point at the inner index. If the angle is less than the current inner angle
    i. set the inner angle equal to this new angle.
    ii. compute the distance from the "beginning" point to the point at the inner index and call it the "current" distance
    iii. compute mid angle as the average of the inner and outer angles
    iv. set the "next" point at the "current" distance from the beginning point along the mid angle

j. (Steps j – m are analogous to steps f – i except they deal with the outer boundary). Check to see if the inner angle is less than or equal to the outer angle. If so, this means that the line has progressed as far as possible down the corridor between the inner and outer boundaries. Therefore:
    i. add the "next" point to the boundary
    ii. set the "beginning" point equal to the "next" point
    iii. move the inner and outer index one past the "next" point
    iv. check to see if the inner index is at the next outer reversal. If so, navigate out of the reversal with these steps:
        1. set the "next" point equal to the midpoint of the points at the inner and outer index
        2. add the "next" point to the boundary
        3. increment the inner and outer indices
        4. set the "next" point equal to the midpoint of the points at the inner and outer index
        5. add the "next" point to the boundary
        6. increment the inner and outer indices.
    v. set the "next" point equal to the midpoint of the points at the inner and outer index
    vi. set the inner and outer angles from the new "beginning" point to the points at the inner and outer indices respectively.

k. Increment the outer index.

l. See if the outer index is at the next outer reversal. If so:
   i. compute the mid angle as the average of the inner and outer angles (adjust inner angle if it not pointing to the inner boundary opposite the outer reversal)
   ii. if the mid angle intersects the centerline of the reversal (see Figure21) then set the "next" point to the intersection
   iii. otherwise, if the mid angle intersects the edge of the reversal (see Figure 21) then set the "next" point to the intersection
   iv. add the "next" point to the boundary
   v. set the "beginning" point equal to the "next" point
   vi. set the inner and outer indices to the point past the reversal
   vii. set the "next" point equal to the midpoint of the points at the inner and outer index
   viii. let the inner and outer angle from the new "beginning" point to the points at the inner and outer indices respectively.

m. Calculate the angle from the "beginning" point to the point at the outer index. If the angle is greater than the current outer angle
   i. set the outer angle equal to this new angle.
   ii. compute the distance from the "beginning" point to the point at the inner index and call it the "current" distance
   iii. compute mid angle as the average of the inner and outer angles
   iv. set the "next" point at the "current" distance from the beginning point along the mid angle.

n. Continue with step C.7.e

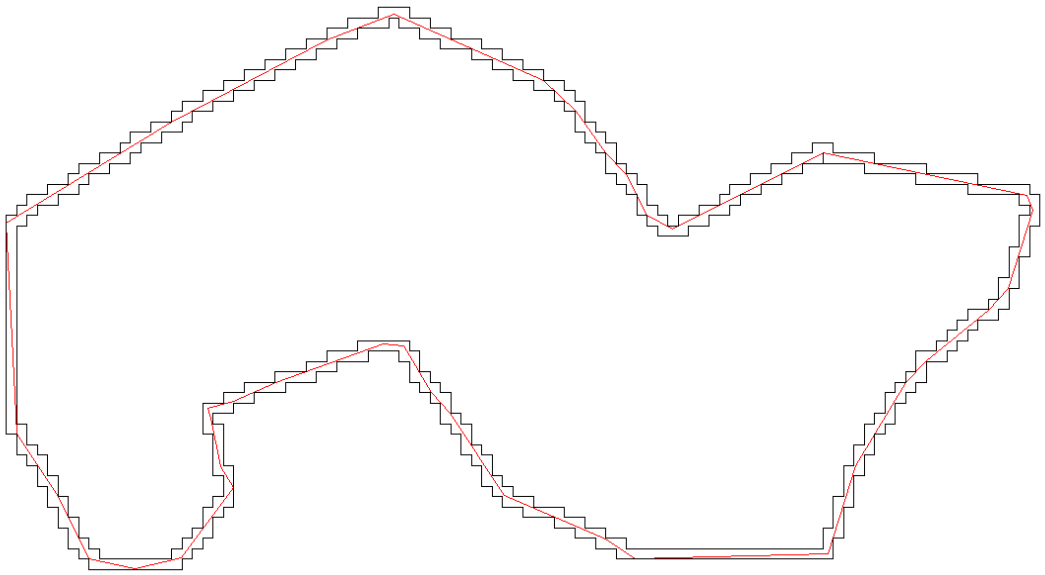

Figure 22. The Final Boundary (in red) with Respect to the Inner and Outer Boundaries

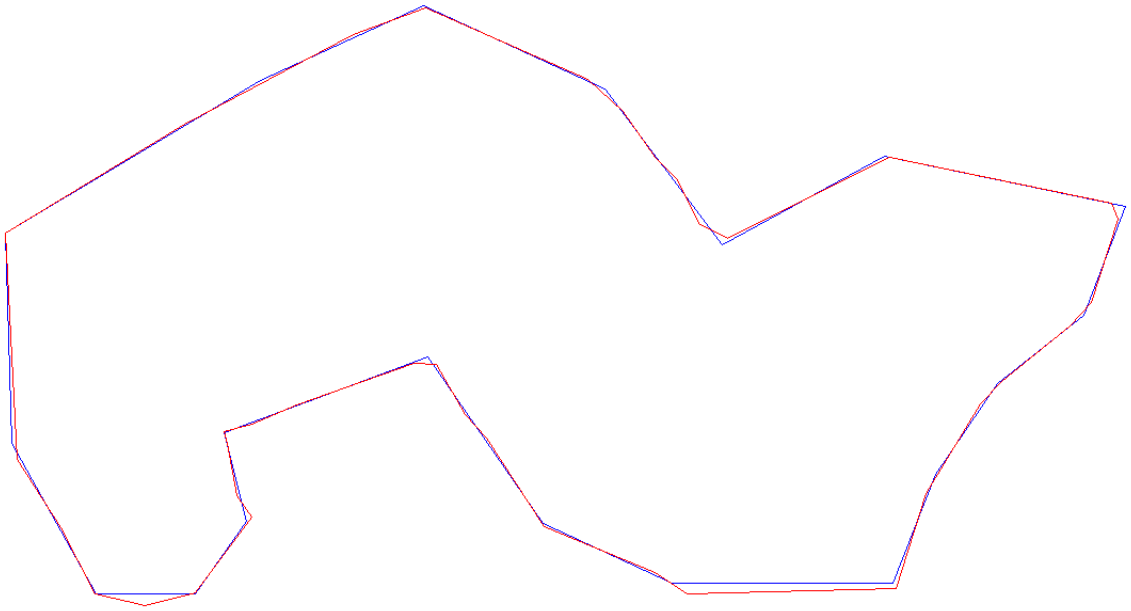

Figure 23. The Final Boundary (in red) with Respect to the Original Boundary (in blue) that Provided the Basis for the Data Grid

3.1 Method C Conclusions

This method does a good job of reproducing the original boundary without the jagged features that would be introduced if the data points were simply connected with line segments. Most errors are introduced at the reversals, so a more intelligent method of determining where to position the point at the reversal could reduce overall divergence.

Appendix A

Selected C++ Source Code for Method A

Appendix A

Selected C++ Source Code for Method A

```
// Automatically delineates spatial areas from gridded data
// arrHot - an array of DPoints, each of which is the center of a grid cell above the threshold
value
// dMinSize - the minimum size for final polygons
// dWidth - horizontal spacing of original grid
// dHeight - vertical spacing of original grid
// Notes: CDPoint is a 3D Point class, CPolyLine is a polygon class that contains a list of
CDPoints
void MethodA(CArray <CDPoint,CDPoint&> &arrHot, double dMinSize, double dWidth, double dHeight)
{
    int iNumHot = arrHot.GetCount();

    if (iNumHot>0) {
        // Group Hot Cells into Clusters
        // Start by sorting array of points by Y Coordinate then X Coordinate
        qsort(arrHot.GetData(), arrHot.GetCount(), sizeof(CDPoint), QCompareDPointYX);
        for (int iHot=0; iHot<iNumHot; iHot++) {
            // Use Z-Coordinate as "Used" flag: -1=unused, 0=used
            arrHot[iHot].m_z = -1.0;
        }

        CArray <CPolyLine, CPolyLine&> arrCluster;
        for (int iHot=0; iHot<iNumHot; iHot++) {
            if (arrHot[iHot].m_z < 0.0) {
                // Start a new Cluster
                CPolyLine cluster(RGB(0,0,0));
                // Add cell point to cluster
                cluster.AddDPoint(arrHot[iHot]);
                // Mark point as being used
                arrHot[iHot].m_z = 0.0;

                // Add to cluster all cells that touch first cell
                MakeCluster(arrHot, cluster, dWidth, dHeight, iHot);

                // Add cluster to the array
                arrCluster.Add(cluster);
            }
        }

        int iNumClusters = arrCluster.GetCount();

        // Convert Each Cluster into a PolyLine
        int iNumAdded = 0;
        for (int iCluster=0; iCluster<iNumClusters; iCluster++) {

            // New polygon
            CPolyLine poly(RGB(0,0,0));

            // Convert Points on Regular Grid to Polygon
            ConvertClusterRegular(arrCluster[iCluster], poly, dWidth, dHeight);

            // Remove acute indentations
            poly.RemoveAcuteIndents();

            // Remove folds
            poly.RemoveFolds();

            if (poly.CalcArea() >= dMinSize) {
                // poly is a complete polygon that passes all checks
                // TODO: do something with it
            }
        }
    }
```

```
}
// Convert cells in cluster to a polyline
void ConvertClusterRegular(const CPolyLine &cluster, CPolyLine &poly, double dWidth, double
dHeight)
{
    // Convert to a regular grid to make conversion simpler
    double dMinX = cluster.GetMinX();
    double dMaxX = cluster.GetMaxX();
    double dXExt = dMaxX - dMinX;
    double dMinY = cluster.GetMinY();
    double dMaxY = cluster.GetMaxY();
    double dYExt = dMaxY - dMinY;
    int iCols = int(dXExt / dWidth + 0.5) + 1;
    int iRows = int(dYExt / dHeight + 0.5) + 1;
    CArray <CDPoint,CDPoint&> arrGrid;
    arrGrid.SetSize(iCols * iRows);

    // Put the Points for the Cluster into the Grid
    int iNumPoints = cluster.NumPoints();
    for (int i=0; i<iNumPoints; i++) {
        CDPoint dPnt = cluster.GetPoint(i);
        int iCol = int((dPnt.m_x - dMinX) / dWidth + 0.5);
        int iRow = iRows - 1 - int((dPnt.m_y - dMinY) / dHeight + 0.5);
        int iCell = iRow * iCols + iCol;
        // Set Grid Cell
        arrGrid[iCell] = CDPoint(dPnt.m_x, dPnt.m_y, 1.0);
    }

    double dWidth2 = dWidth / 2.0;
    double dHeight2 = dHeight / 2.0;

    // Fill in gaps in grid where cells are joined only by a corner
    // As this may sometimes lead to an inner loop or a missing section
    BOOL bDone = FALSE;
    while (!bDone) {
        bDone = TRUE;
        for (int iRow = 0; iRow<iRows-1; iRow++) {
            for (int iCol = 0; iCol<iCols-1; iCol++) {
                int iCur = iRow * iCols + iCol;
                int iRt = iCur + 1;
                int iDn = iCur + iCols;
                int iOp = iCur + iCols + 1;
                if (arrGrid[iCur].m_z > 0.0 && arrGrid[iOp].m_z > 0.0 && arrGrid[iRt].m_z == 0.0
&& arrGrid[iDn].m_z == 0.0) {
                        CDPoint dPnt;
                        dPnt.m_x = arrGrid[iOp].m_x;
                        dPnt.m_y = arrGrid[iCur].m_y;
                        dPnt.m_z = 1.0;
                        arrGrid[iRt] = dPnt;
                        bDone = FALSE; // Redo in case we just added another
                } else if (arrGrid[iCur].m_z == 0.0 && arrGrid[iOp].m_z == 0.0 && arrGrid[iRt].m_z
> 0.0 && arrGrid[iDn].m_z > 0.0) {
                        CDPoint dPnt;
                        dPnt.m_x = arrGrid[iDn].m_x;
                        dPnt.m_y = arrGrid[iRt].m_y;
                        dPnt.m_z = 1.0;
                        arrGrid[iCur] = dPnt;
                        bDone = FALSE; // Redo in case we just added another
                }
            }
        }
    }

    // Make a list of all segments where cells adjoin blank cells
    CArray <CDPoint,CDPoint&> arrVert;
    for (int iRow = 0; iRow<iRows; iRow++) {
        for (int iCol = 0; iCol<iCols; iCol++) {
            int iCur = iRow * iCols + iCol;
            CDPoint dPnt = arrGrid[iCur];
            if (dPnt.m_z > 0.0) {
                BOOL bUp = FALSE;
```

```
            if (iRow>0) {
                    bUp = (arrGrid[iCur-iCols].m_z > 0.0);
            }
            BOOL bRt = FALSE;
            if (iCol<iCols-1) {
                    bRt = (arrGrid[iCur+1].m_z > 0.0);
            }
            BOOL bDn = FALSE;
            if (iRow<iRows-1) {
                    bDn = (arrGrid[iCur+iCols].m_z > 0.0);
            }
            BOOL bLf = FALSE;
            if (iCol>0) {
                    bLf = (arrGrid[iCur-1].m_z > 0.0);
            }
            if (!bUp) {
                    arrVert.Add(CDPoint(dPnt.m_x-dWidth2, dPnt.m_y+dHeight2));
                    arrVert.Add(CDPoint(dPnt.m_x+dWidth2, dPnt.m_y+dHeight2));
            }
            if (!bRt) {
                    arrVert.Add(CDPoint(dPnt.m_x+dWidth2, dPnt.m_y+dHeight2));
                    arrVert.Add(CDPoint(dPnt.m_x+dWidth2, dPnt.m_y-dHeight2));
            }
            if (!bDn) {
                    arrVert.Add(CDPoint(dPnt.m_x+dWidth2, dPnt.m_y-dHeight2));
                    arrVert.Add(CDPoint(dPnt.m_x-dWidth2, dPnt.m_y-dHeight2));
            }
            if (!bLf) {
                    arrVert.Add(CDPoint(dPnt.m_x-dWidth2, dPnt.m_y-dHeight2));
                    arrVert.Add(CDPoint(dPnt.m_x-dWidth2, dPnt.m_y+dHeight2));
            }
        }
    }
}

// Add First Segment to Polygon
CDPoint dFirst = arrVert[0];
poly.AddDPoint(dFirst);
CDPoint dLast = arrVert[1];
poly.AddDPoint(dLast);

// Setup a list of segment indices
int iNumSegs = arrVert.GetCount()/2 - 1;
CArray <BOOL,BOOL> arrSeg;
arrSeg.SetSize(iNumSegs);
for (int i=0; i<iNumSegs; i++) {
    arrSeg[i] = i+1;
}

// Add unused Segments that attach until we come back to beginning
while (dFirst != dLast) {
    for (int iSeg=0; iSeg<iNumSegs; iSeg++) {
        int iIndex = arrSeg[iSeg]*2;
        if (dLast.Compare(arrVert[iIndex], 1e-10)) {
            dLast = arrVert[iIndex+1];
            poly.AddDPoint(dLast);
            arrSeg.RemoveAt(iSeg);
            iNumSegs--;
            break;
        } else if (dLast.Compare(arrVert[iIndex], 1e-10)) {
            dLast = arrVert[iIndex];
            poly.AddDPoint(dLast);
            arrSeg.RemoveAt(iSeg);
            iNumSegs--;
            break;
        }
    }
}

// Finally, Remove In-Line Points
poly.RemoveInLine();
```

```
}

// Removes all indentation <= 90 degrees
// Note:  m_arrPoint contains list of CDPoints for polygon
int CPolyLine::RemoveAcuteIndents(double dMaxDegrees)
{
    double dMaxAngle = dMaxDegrees / 180.0 * PI;

    BOOL bDone = FALSE;
    while (!bDone) {
        bDone = TRUE;
        for (int i=0; i<m_arrPoint.GetCount(); i++) {
            // Check angle of next two segments
            int j = (i+1) % m_arrPoint.GetCount();
            int k = (j+1) % m_arrPoint.GetCount();
            CDPoint dPnt1 = m_arrPoint[i];
            CDPoint dPnt2 = m_arrPoint[j];
            CDPoint dPnt3 = m_arrPoint[k];
            double dAngle = CalcAngle3D(dPnt2, dPnt1, dPnt3);
            // <= 90 degrees?
            if (dAngle <= dMaxAngle) {
                // Yes, see if segment from Pnt1 to Pnt3 would be inside or outside of existing
polygon
                CDPoint dChk((dPnt1.m_x+dPnt3.m_x)/2.0, (dPnt1.m_y+dPnt3.m_y)/2.0);
                if (!IsInside(dChk)) {
                        // Outside
                        // Now, check to see if Triangle (dPnt1,dPnt2,dPnt3) encloses any other
point
                        BOOL bEncloses = FALSE;
                        for (int m=0; m<m_arrPoint.GetCount(); m++) {
                                if (m!=i && m!=j && m!=k) {
                                        if (InTriangle(dPnt1, dPnt2, dPnt3, m_arrPoint[m])) {
                                                bEncloses = TRUE;
                                                break;
                                        }
                                }
                        }
                        // No, go ahead and remove point 2 to make a short-cut from Pnt1 to Pnt3
                        if (!bEncloses) {
                                m_arrPoint.RemoveAt(j);
                                bDone = FALSE; // Will need to make another pass
                        }
                }
            }
        }
        if (!bDone) {
            RemoveInLine(1e-12);
        }
    }
    ResetArea();                        // Reset Area
    return m_arrPoint.GetCount();
}

// Removes segments that fold-back on itself, and returns number of point remaining
int CPolyLine::RemoveFolds()
{
    int iSize = m_arrPoint.GetSize();
    for (int i=0; i<iSize && iSize>2; i++) {
        int j = (i+iSize+0) % iSize;  // This Point
        int k = (i+iSize+1) % iSize;  // Next Point
        int l = (i+iSize+2) % iSize;  // Point After Next Point
        if (m_arrPoint[j] == m_arrPoint[l]) {
            // Remove 2 Points that Make Segment
            if (l > k) {
                m_arrPoint.RemoveAt(l);
                m_arrPoint.RemoveAt(k);
                iSize = m_arrPoint.GetSize();
            } else {
                m_arrPoint.RemoveAt(k);
                m_arrPoint.RemoveAt(l);
                iSize = m_arrPoint.GetSize();
```

```
            }
            i -= 2;     // Back Up and Try Previous Segment
        }
    }

    for (int i=0; i<iSize && iSize>2; i++) {
        int j = (i+iSize+0) % iSize;  // This Point
        int k = (i+iSize+1) % iSize;  // Next Point
        int l = (i+iSize+2) % iSize;  // Point After Next Point

        // Check for Co-Linear Points
        if (PointLineDist(m_arrPoint[j], m_arrPoint[k], m_arrPoint[l]) < 1e-8 ) {
            // Doubles back across itself, Remove center point
            m_arrPoint.RemoveAt(k);
            iSize = m_arrPoint.GetSize();
        }
    }

    return m_arrPoint.GetSize();
}
```

Appendix B

Selected C++ Source Code for Method B

Appendix B

Selected C++ Source Code for Method B

```
// Automatically delineates spatial areas around markers of interest
// arrMarker - an array of DPoints, each of which is a point of interest
// dGridSize – size of box to create around markers
// dMinSize - the minimum size for final polygons
void MethodB(CArray <CDPoint,CDPoint&> arrMarker, double dGridSize, double dMinSize)
{
    double dGridSize2 = dGridSize / 2.0;
    int iNumMarkers = arrMarker.GetCount();
    if (iNumMarkers<=0) {
        return;
    }

    // Sort Markers by Y then X
    qsort(arrMarker.GetData(), arrMarker.GetCount(), sizeof(CDPoint), QCompareDPointYX);

    // Create an Array of PolyLines (Boxes)
    CArray <CPolyLine,CPolyLine&> arrPoly;
    arrPoly.SetSize(iNumMarkers);
    for (int iMarker=0; iMarker<iNumMarkers; iMarker++) {
        CPolyLine poly(RGB(0,0,0));
        CDPoint dCenter = arrMarker[iMarker];
        poly.AddDPoint(CDPoint(dCenter.m_x-dGridSize2, dCenter.m_y+dGridSize2));
        poly.AddDPoint(CDPoint(dCenter.m_x+dGridSize2, dCenter.m_y+dGridSize2));
        poly.AddDPoint(CDPoint(dCenter.m_x+dGridSize2, dCenter.m_y-dGridSize2));
        poly.AddDPoint(CDPoint(dCenter.m_x-dGridSize2, dCenter.m_y-dGridSize2));
        poly.AddDPoint(CDPoint(dCenter.m_x-dGridSize2, dCenter.m_y+dGridSize2));
        poly.SetExtents();
        poly.m_bSelect = FALSE;
        arrPoly[iMarker] = poly;
    }

    // Group all the PolyLines into Clusters
    CArray <CPolyLine, CPolyLine&> arrCluster;
    for (int iMarker=0; iMarker<iNumMarkers; iMarker++) {
        if (!arrPoly[iMarker].m_bSelect) {
            // Start a new Cluster
            arrPoly[iMarker].m_bSelect = TRUE;              // Mark as being used
            CPolyLine cluster = arrPoly[iMarker];

            // Add to cluster all markers that touch first marker
            TargetCluster(arrPoly, cluster, iMarker);

            // Add cluster to array
            arrCluster.Add(cluster);
        }
    }

    int iNumClusters = arrCluster.GetCount();

    int iNumAdded = 0;
    for (int iCluster=0; iCluster<iNumClusters; iCluster++) {
        arrCluster[iCluster].RemoveAcuteIndents(110.0);     // Cleanup Edges
        arrCluster[iCluster].SetExtents();    // Update the Stored Extents
        if (arrCluster[iCluster].CalcArea() >= dMinSize) {   // Check Size
            // arrCluster[iCluster] is a complete polygon that passes all checks
            // TODO: do something with it
        }
    }
}

// Group all the marker polygons (arrMarker)
```

```
// that touch the given marker (iMarker)
// into a single polygon (cluster)
void TargetCluster(CArray <CPolyLine,CPolyLine&> &arrMarker, CPolyLine &cluster, int iMarker)
{
    int iNumMarker = arrMarker.GetCount();

    CArray <CPolyLine,CPolyLine&> arrNew;

    CArray <int,int> arrStack;// Save Marker to Check
    arrStack.Add(iMarker);

    while (arrStack.GetCount()>0) {   // Any Markers to Check?
        // Yes
        int iStack = arrStack.GetCount()-1;
        iMarker = arrStack[iStack];
        arrStack.RemoveAt(iStack);
        CPolyLine &Marker = arrMarker[iMarker];

        // Check forward in list (upward in space)
        for (int i=iMarker; i<iNumMarker; i++) {
            // See if checked far enough
            if (Marker.GetMaxY() < arrMarker[i].GetMinY()) {
                break;
            }
            if (!arrMarker[i].m_bSelect) {
                // Found Unused Marker
                // Merge Two Polygons, resulting Polygons go into arrNew array
                if (Union(cluster, arrMarker[i], arrNew, TRUE)) {
                        // Choose Largest PolyLine, assume others are holes
                        int iNew = arrNew.GetCount();
                        int iLrg = 0;
                        double dLrg = 0.0;
                        for (int j=0; j<iNew; j++) {
                                if (dLrg < arrNew[j].CalcArea()) {
                                        dLrg = arrNew[j].CalcArea();
                                        iLrg = j;
                                }
                        }
                        cluster = arrNew[iLrg];
                        // Remove colinear points
                        cluster.RemoveInLine();
                        // Mark as used
                        arrMarker[i].m_bSelect = TRUE;
                        // Save marker on stack so we can check everwhere from here
                        arrStack.Add(i);
                }
            }
        }

        // Check backward in list (downward in space)
        for (int i=iMarker; i>=0; i--) {
            // See if checked far enough
            if (Marker.GetMinY() > arrMarker[i].GetMaxY()) {
                break;
            }
            if (!arrMarker[i].m_bSelect) {
                // Found Unused Marker
                // Merge Two Polygons, resulting Polygons go into arrNew array
                if (Union(cluster, arrMarker[i], arrNew, TRUE)) {
                        // Choose Largest PolyLine, assume others are holes
                        int iNew = arrNew.GetCount();
                        int iLrg = 0;
                        double dLrg = 0.0;
                        for (int j=0; j<iNew; j++) {
                                if (dLrg < arrNew[j].CalcArea()) {
                                        dLrg = arrNew[j].CalcArea();
                                        iLrg = j;
                                }
                        }
                        cluster = arrNew[iLrg];
                        // Remove colinear points
```

```
                        cluster.RemoveInLine();
                        // Mark as used
                        arrMarker[i].m_bSelect = TRUE;
                        // Save marker on stack so we can check everwhere from here
                        arrStack.Add(i);
                    }
                }
            }
        }
    }
```

Appendix C

Selected C++ Source Code for Method C

Appendix C

Selected C++ Source Code for Method C

```
CPolyLine* Grid2Poly(const CArray <double,double> &arrData, int iCols, int iRows, double dXMin,
double dYMin, double dGridSize)
{
    if (iCols>1 && iRows>1 && arrData.GetCount()==iRows*iCols) {
        // Max and Min Z at edge
        double dMinZ = -VALUE_BLANK;
        double dMaxZ = VALUE_BLANK;

        // Make a local copy
        CArray <double,double> arrTemp;
        arrTemp.SetSize(arrData.GetCount());
        for (int i=0; i<arrData.GetCount(); i++) {
            arrTemp[i] = arrData[i];
        }

        // Check first row with data and see if one point stands alone
        BOOL bFound = FALSE;
        for (int iRow=0; iRow<iRows && !bFound; iRow++) {
            for (int iCol=1; iCol<iCols-1; iCol++) {
                int iIndex = iRow * iCols + iCol;
                if (arrTemp[iIndex] > VALUE_BLANK) {
                        bFound = TRUE; // Found data
                        if (arrTemp[iIndex-1] <= VALUE_BLANK &&
                        arrTemp[iIndex+1] <= VALUE_BLANK) {
                                // Stands alone, get rid of it
                                arrTemp[iIndex] = VALUE_BLANK;
                                break;
                        }
                }
            }
        }

        // Find all the non-blank points that are adjacent to a blank point
        CArray <CPoint,CPoint&> arrPoints;
        int iIndex = 0;
        for (int iRow=0; iRow<iRows; iRow++) {
            for (int iCol=0; iCol<iCols; iCol++) {
                if (arrTemp[iIndex] > VALUE_BLANK) {
                        BOOL bEdge = (iRow==0 || iCol==0 || iRow==iRows-1 || iCol==iCols-1);
                        if (bEdge) {
                                arrPoints.Add(CPoint(iCol,iRow));
                                if (dMinZ > arrTemp[iIndex]) {
                                        dMinZ = arrTemp[iIndex];
                                }
                                if (dMaxZ < arrTemp[iIndex]) {
                                        dMaxZ = arrTemp[iIndex];
                                }
                        } else {
                        if (arrTemp[iIndex-1]<=VALUE_BLANK || arrTemp[iIndex+1]<=VALUE_BLANK ||
                            arrTemp[iIndex-iCols]<=VALUE_BLANK ||
                            arrTemp[iIndex+iCols]<=VALUE_BLANK ||
                                        arrTemp[iIndex-iCols-1]<=VALUE_BLANK ||
                            arrTemp[iIndex-iCols+1]<=VALUE_BLANK ||
                            arrTemp[iIndex+iCols-1]<=VALUE_BLANK ||
                            arrTemp[iIndex+iCols+1]<=VALUE_BLANK) {
                                        arrPoints.Add(CPoint(iCol,iRow));
                                        if (dMinZ > arrTemp[iIndex]) {
                                                dMinZ = arrTemp[iIndex];
                                        }
                                        if (dMaxZ < arrTemp[iIndex]) {
                                                dMaxZ = arrTemp[iIndex];
                                        }
                                }
```

```
                }
            }
            iIndex++;
        }
    }

    // Decide on edge Z-Offset
    double dEdgeZ = dMinZ;
    if (fabs(dMaxZ) < fabs(dMinZ)) {
        dEdgeZ = dMaxZ;
    }
    dEdgeZ = Round(dEdgeZ, 2);

    // This is the inner polygon
    CPolyLine *pInner = Points2Poly(arrPoints, dXMin, dYMin, dGridSize);
    if (pInner) {
        if (pInner->NumPoints() <= 5) {
            // Inner polygon is probably rectangle, can just return it
            pInner->SetZOffset(dEdgeZ);
            return pInner;
        }
        arrPoints.RemoveAll();
        for (int iRow=-1; iRow<=iRows; iRow++) {
            for (int iCol=-1; iCol<=iCols; iCol++) {
                int iIndex = iRow * iCols + iCol;
                if (iRow>=0 && iRow<iRows && iCol>=0 && iCol<iCols) {
                    // Within the grid
                    if (arrTemp[iIndex] <= VALUE_BLANK) {
                        if (iCol>0 && arrTemp[iIndex-1]>VALUE_BLANK ||
                            iCol<iCols-1 && arrTemp[iIndex+1]>VALUE_BLANK ||
                            iRow>0 && arrTemp[iIndex-iCols]>VALUE_BLANK ||
                            iRow<iRows-1 && arrTemp[iIndex+iCols]>VALUE_BLANK ||
                            iRow>0 && iCol>0 && arrTemp[iIndex-iCols-
1]>VALUE_BLANK ||
                            iRow>0 && iCol<iCols-1 && arrTemp[iIndex-iCols+1] > VALUE_BLANK ||
                            iRow<iRows-1 && iCol>0 && arrTemp[iIndex+iCols-1] > VALUE_BLANK ||
                            iRow<iRows-1 && iCol<iCols-1 && arrTemp[iIndex+iCols+1] >
                            VALUE_BLANK) {
                            arrPoints.Add(CPoint(iCol,iRow));
                        }
                    }
                } else {
                    // Outside the grid
                    if (iCol>0 && iRow>=0 && iRow<iRows && arrTemp[iIndex-1] > VALUE_BLANK ||
                        iCol<iCols-1 && iRow>=0 && iRow<iRows && arrTemp[iIndex+1] >
                        VALUE_BLANK ||
                        iRow>0 && iCol>=0 && iCol<iCols && arrTemp[iIndex-iCols] > VALUE_BLANK
                        ||
                        iRow<iRows-1 && iCol>=0 && iCol<iCols && arrTemp[iIndex+iCols] >
                        VALUE_BLANK ||
                            iRow>0 && iCol>0 && arrTemp[iIndex-iCols-1] > VALUE_BLANK
||
                        iRow>0 && iCol<iCols-1 && arrTemp[iIndex-iCols+1] > VALUE_BLANK ||
                        iRow<iRows-1 && iCol>0 && arrTemp[iIndex+iCols-1] > VALUE_BLANK ||
                        iRow<iRows-1 && iCol<iCols-1 && arrTemp[iIndex+iCols+1] > VALUE_BLANK)
                        {
                            arrPoints.Add(CPoint(iCol,iRow));
                        }
                    }
                }
            }

            // This is the outer polygon
            CPolyLine *pOuter = Points2Poly(arrPoints, dXMin, dYMin, dGridSize);
            if (pOuter) {
                CArray <CDPoint,CDPoint&> arrInner;
                pInner->GetPoints(arrInner);
                int iNumInner = arrInner.GetCount();
                CArray <CDPoint,CDPoint&> arrOuter;
                pOuter->GetPoints(arrOuter);
                int iNumOuter = arrOuter.GetCount();
```

```
if (iNumOuter > 3 && iNumOuter == iNumInner &&
!pOuter->ComparePoints(pInner,FALSE,1e-6)) {
	// Find all the reversal points
	CArray <int,int> arrRevIn;
	CArray <int,int> arrRevOut;
	for (int i=1; i<iNumInner-2; i++) {
	if (!arrOuter[i-1].Compare(arrOuter[i],1e-6) &&
	!arrOuter[i].Compare(arrOuter[i+1],1e-6) &&
	!arrOuter[i+1].Compare(arrOuter[i+2],1e-6)) {
			double dAngle1 = Angle(arrOuter[i-1], arrOuter[i]);
			double dAngle2 = Angle(arrOuter[i], arrOuter[i+1]);
			double dAngle3 = Angle(arrOuter[i+1], arrOuter[i+2]);
			if (fabs(dAngle1-dAngle3) > 1e-6) {
				// Hit Reversal
				double dTurn = NormalizeAngle(dAngle2 - dAngle1);
				if (fabs(dTurn-PI/2.0) < 1e-6) {
					// Left turn - Outer reversal
					arrRevOut.Add(i);
				}
			}
		}
	if (!arrInner[i-1].Compare(arrInner[i],1e-6) &&
	!arrInner[i].Compare(arrInner[i+1],1e-6) &&
	!arrInner[i+1].Compare(arrInner[i+2],1e-6)) {
			double dAngle1 = Angle(arrInner[i-1], arrInner[i]);
			double dAngle2 = Angle(arrInner[i], arrInner[i+1]);
			double dAngle3 = Angle(arrInner[i+1], arrInner[i+2]);
			if (fabs(dAngle1-dAngle3) > 1e-6) {
				// Hit Reversal
				double dTurn = NormalizeAngle(dAngle2 - dAngle1);
				if (fabs(dTurn-PI/2.0) >= 1e-6) {
					// Right turn - Inner reversal
					arrRevIn.Add(i);
				}
			}
		}
	}
	// Add another entry to keep from running off end of array
	arrRevIn.Add(-1);
	arrRevOut.Add(-1);
	int iNextRevOut = 0;
	int iNextRevIn = 0;

	CArray <CDPoint,CDPoint&> arrNew;
	// Start at bottom of Outer
	CDPoint dPntBeg = MidPoint(arrOuter[0], arrOuter[1]);
	arrNew.Add(dPntBeg);

	int iInner = 1;
	int iOuter = 1;
	double dAngleInner = NormalizeAngle(Angle(dPntBeg, arrInner[iInner]));
	double dAngleOuter = NormalizeAngle(Angle(dPntBeg, arrOuter[iOuter]));
	double dAngle = 0.0;
	double dAngleMid = 0.0;

	CDPoint dPntNext = MidPoint(arrInner[iInner], arrOuter[iOuter]);
	int iNext = iInner;

	BOOL bDone = FALSE;
	while (iInner < iNumInner-1) {

		if (dAngleInner <= dAngleOuter) {
			// Angles crossed, so output best point
			arrNew.Add(dPntNext);
			// and start again from here
			dPntBeg = dPntNext;
			iNext++;
			if (iNext >= iNumInner) {
				break;
			}
```

```
                                        iInner = iNext;
                                        iOuter = iNext;
                                        if (iInner == arrRevIn[iNextRevIn]) {
                                                // Already in a reversal, so navigate out
                                                iNextRevIn++;
                                                dPntNext = MidPoint(arrInner[iInner],
arrOuter[iOuter]);
                                                arrNew.Add(dPntNext);
                                                iNext++;
                                                iInner = iNext;
                                                iOuter = iNext;
                                                dPntNext = MidPoint(arrInner[iInner],
arrOuter[iOuter]);
                                                arrNew.Add(dPntNext);
                                                dPntBeg = dPntNext;
                                                iNext++;
                                                iInner = iNext;
                                                iOuter = iNext;
                                        }
                                        dAngleInner = NormalizeAngle(Angle(dPntBeg,
arrInner[iInner]));
                                        dAngleOuter = NormalizeAngle(Angle(dPntBeg,
arrOuter[iOuter]));
                                        dPntNext = MidPoint(arrInner[iInner], arrOuter[iOuter]);
                                        if (dAngleInner < dAngleOuter) {
                                                // De-Normalize
                                                if (fabs(dAngleOuter-dAngleMid) >
                                    fabs(dAngleInner-dAngleMid)) {
                                                        dAngleOuter -= PI*2.0;
                                                } else {
                                                        dAngleInner += PI*2.0;
                                                }
                                        }
                                    }

                                    iInner++;

                                    if (iInner == arrRevIn[iNextRevIn]) {
                                            // Hit Reversal
                                            iNextRevIn++;

                                            // Need to check latest outer angle
                                            dAngle = NormalizeAngle(Angle(dPntBeg, arrOuter[iInner]));
                                            if (dAngleOuter < dAngle) {
                                                    dAngleOuter = dAngle;
                                            }

                                            // Extend current line out past edge
                                            dAngleMid = (dAngleInner + dAngleOuter) / 2.0;
                                            double dDist = LineLength(dPntBeg, arrInner[iInner+1]);
                                CDPoint dPntEnd = CDPoint(dPntBeg.m_x + cos(dAngleMid)*dDist,
                                dPntBeg.m_y + sin(dAngleMid)*dDist);

                                            // Find intersection point with center of reversal
                                CDPoint dPntChk1 = MidPoint(arrInner[iInner], arrInner[iInner+1]);
                                CDPoint dPntChk2 = MidPoint(arrOuter[iInner], arrOuter[iInner+1]);
                                if (LineLine(dPntBeg, dPntEnd, dPntChk1, dPntChk2, dPntNext, TRUE)) {
                                                    // Found
                                                    arrNew.Add(dPntNext);
                                            } else {
                                                    // Not found, try intersection with edge
                                                if (LineLine(dPntBeg, dPntEnd, arrInner[iInner],
                                                arrInner[iInner+1], dPntNext, FALSE)) {
                                                        arrNew.Add(dPntNext);
                                                } else {
                                                        break;  // Must be an error
                                                }
                                            }
                                            dPntBeg = dPntNext;
                                            iNext = iInner + 1;
                                            iOuter = iNext;
```

```
                                        iInner = iNext;
                                        dAngleInner = NormalizeAngle(Angle(dPntBeg,
arrInner[iInner]));
                                        dAngleOuter = NormalizeAngle(Angle(dPntBeg,
arrOuter[iOuter]));
                                        dPntNext = MidPoint(arrInner[iInner], arrOuter[iOuter]);
                                }

                                if (iInner < iNumInner) {
                                        // Need to move angle in?
                                        dAngle = NormalizeAngle(Angle(dPntBeg, arrInner[iInner]));
                                        if (dAngleInner > dAngle) {
                                                // Yes
                                                dAngleInner = dAngle;
                                                if (dAngleInner > dAngleOuter) {
                                                        // This point is inside,
                                   // so keep it until a better point comes around
                                                        double dDist = LineLength(dPntBeg,
arrInner[iInner]);
                                                        dAngleMid = (dAngleInner + dAngleOuter) /
2.0;
                                    dPntNext = CDPoint(dPntBeg.m_x + cos(dAngleMid)*dDist,
                                    dPntBeg.m_y + sin(dAngleMid)*dDist);
                                                        iNext = iInner;
                                                }
                                        }
                                }

                                if (dAngleInner <= dAngleOuter) {
                                        // Angles crossed, so output best point
                                        arrNew.Add(dPntNext);
                                        // Start again from here
                                        dPntBeg = dPntNext;
                                        iNext++;
                                        if (iNext >= iNumInner) {
                                                break;
                                        }
                                        iInner = iNext;
                                        iOuter = iNext;
                                        if (iOuter == arrRevOut[iNextRevOut]) {
                                                // Already in a reversal, so navigate out
                                                iNextRevOut++;
                                                dPntNext = MidPoint(arrInner[iInner],
arrOuter[iOuter]);
                                                arrNew.Add(dPntNext);
                                                iNext++;
                                                iInner = iNext;
                                                iOuter = iNext;
                                                dPntNext = MidPoint(arrInner[iInner],
arrOuter[iOuter]);
                                                arrNew.Add(dPntNext);
                                                dPntBeg = dPntNext;
                                                iNext++;
                                                iInner = iNext;
                                                iOuter = iNext;
                                        }
                                        dAngleInner = NormalizeAngle(Angle(dPntBeg,
arrInner[iInner]));
                                        dAngleOuter = NormalizeAngle(Angle(dPntBeg,
arrOuter[iOuter]));
                                        dPntNext = MidPoint(arrInner[iInner], arrOuter[iOuter]);
                                        if (dAngleInner < dAngleOuter) {
                                                // De-Normalize
                                                if (fabs(dAngleOuter-dAngleMid) >
                                fabs(dAngleInner-dAngleMid)) {
                                                        dAngleOuter -= PI*2.0;
                                                } else {
                                                        dAngleInner += PI*2.0;
                                                }
                                        }
                                }
```

```
                iOuter++;

                if (iOuter == arrRevOut[iNextRevOut]) {
                        // Hit Reversal
                        iNextRevOut++;

                        // Need to check latest inner angle
                        dAngle = NormalizeAngle(Angle(dPntBeg, arrInner[iOuter]));
                        if (dAngleInner > dAngle) {
                                dAngleInner = dAngle;
                        }

                        // Extend current line out past edge
                        dAngleMid = (dAngleInner + dAngleOuter) / 2.0;
                        double dDist = LineLength(dPntBeg, arrOuter[iOuter+1]);
                CDPoint dPntEnd = CDPoint(dPntBeg.m_x + cos(dAngleMid)*dDist,
                dPntBeg.m_y + sin(dAngleMid)*dDist);

                        // Find intersection point with center of reversal
                CDPoint dPntChk1 = MidPoint(arrOuter[iOuter], arrOuter[iOuter+1]);
                CDPoint dPntChk2 = MidPoint(arrInner[iOuter], arrInner[iOuter+1]);
                if (LineLine(dPntBeg, dPntEnd, dPntChk1, dPntChk2, dPntNext, TRUE)) {
                                // Found
                                arrNew.Add(dPntNext);
                        } else {
                                // Not found, try intersection with edge
                    if (LineLine(dPntBeg, dPntEnd, arrOuter[iOuter],
                    arrOuter[iOuter+1], dPntNext, FALSE)) {
                                        arrNew.Add(dPntNext);
                                } else {
                                        break;  // Must be an error
                                }
                        }
                        // Start again from here
                        dPntBeg = dPntNext;
                        iNext = iOuter + 1;
                        iOuter = iNext;
                        iInner = iNext;
                        dAngleInner = NormalizeAngle(Angle(dPntBeg,
arrInner[iInner]));
                        dAngleOuter = NormalizeAngle(Angle(dPntBeg,
arrOuter[iOuter]));
                        dPntNext = MidPoint(arrInner[iInner], arrOuter[iOuter]);
                }

                if (iOuter < iNumOuter) {
                        // Need to move angle in?
                        dAngle = NormalizeAngle(Angle(dPntBeg, arrOuter[iOuter]));
                        if (dAngleOuter < dAngle) {
                                // Yes
                                dAngleOuter = dAngle;
                                if (dAngleInner > dAngleOuter) {
                                        // This point is inside,
                    // so keep it until a better point comes around
                                        double dDist = LineLength(dPntBeg,
arrOuter[iOuter]);
                                        dAngleMid = (dAngleInner + dAngleOuter) /
2.0;
                    dPntNext = CDPoint(dPntBeg.m_x + cos(dAngleMid)*dDist,
                    dPntBeg.m_y + sin(dAngleMid)*dDist);
                                        iNext = iOuter;
                                }
                        }
                }
        }

        if (dPntNext != dPntBeg) {
                arrNew.Add(dPntNext);
        }
```

```
                    CPolyLine* pNew = new CPolyLine(RGB(0,0,0));
                    for (int i=0; i<arrNew.GetCount(); i++) {
                        pNew->AddDPoint(arrNew[i]);
                    }
                    delete pOuter;
                    delete pInner;
                    pNew->RemoveDups();
                    pNew->ClosePoly();
                    pNew->SetZOffset(dEdgeZ);
                    return pNew;
                }
                delete pInner;
                pOuter->SetZOffset(dEdgeZ);
                return pOuter;
            }
            pInner->SetZOffset(dEdgeZ);
            return pInner;
        }
    }
    return NULL;
}

// Convert list of indices on edge to a polygon
CPolyLine* Points2Poly(CArray <CPoint,CPoint&> &arrPoints, double dXMin, double dYMin, double
dGridSize)
{
    CPolyLine poly(RGB(0,0,0));
    CArray <int,int> arrOrder;
    while (ConnectPoints(arrPoints, arrOrder)) {
        // Transfer points to a new PolyLine
        CPolyLine newPoly(RGB(0,0,0));
        for (int i=0; i<arrOrder.GetCount(); i++) {
            int iPoint = arrOrder[i];
            // Use insert so we can get duplicate points
            newPoly.InsertDPoint(
            CDPoint(double(arrPoints[iPoint].x) * dGridSize + dXMin, double(arrPoints[iPoint].y) *
            dGridSize + dYMin),
            newPoly.NumPoints());
        }

        // Mark Used Points
        // (need to do it this way because points can be used more than once)
        for (int i=0; i<arrOrder.GetCount(); i++) {
            int iPoint = arrOrder[i];
            arrPoints[iPoint] = CPoint(-2,-2);
        }

        // Remove Used Points
        for (int i=arrPoints.GetCount()-1; i>=0; i--) {
            if (arrPoints[i] == CPoint(-2,-2)) {
                arrPoints.RemoveAt(i);
            }
        }

        // Cleanup PolyLine
        newPoly.RemoveInLine(1e-6, TRUE);
        newPoly.ClosePoly();

        // See if this PolyLine is inside a previous one
        if (!poly.IsInside(&newPoly)) {
            // No, keep this one
            poly = newPoly;
        }
    }

    if (poly.NumPoints() > 3) {
        // Make a new PolyLine on the heap and return it
        CPolyLine *pLine = new CPolyLine(poly);
        pLine->SetExtents();
        return pLine;
    }
```

```
    return NULL;
}

// Connect index points together to make a continuous polygon
// Order of points are returned in arrOrder
BOOL ConnectPoints(const CArray<CPoint,CPoint&> &arrPoints, CArray<int,int> &arrOrder)
{
    arrOrder.RemoveAll();
    int iNumPoints = arrPoints.GetCount();
    if (iNumPoints >= 8) {

        // Setup list of available points
        CArray <int,int> arrAvail;
        arrAvail.SetSize(iNumPoints);
        for (int i=0; i<iNumPoints; i++) {
            arrAvail[i] = i;
        }

        int iCurrent = 0;
        arrOrder.Add(iCurrent);
        arrAvail.RemoveAt(iCurrent);
        CPoint pntCurrent = arrPoints[iCurrent];

        while (arrAvail.GetCount() > 0) {
            int iMinDistance = iNumPoints;
            int iClosest = -1;
            int iClosestIndex = -1;
            CArray <int,int> arrTies;
            for (int i=0; i<arrAvail.GetCount(); i++) {
                int iCheck = arrAvail[i];
                int iDistance =
                abs(pntCurrent.x - arrPoints[iCheck].x) +
                abs(pntCurrent.y - arrPoints[iCheck].y);
                if (iMinDistance > iDistance) {
                        iMinDistance = iDistance;     // Taxi distance to point
                        iClosest = iCheck;                    // Index into arrPoints
                        iClosestIndex = i;                    // Index into arrAvail
                        arrTies.SetSize(1);           // Reset arrTies
                        arrTies[0] = i;
                } else if (iMinDistance == iDistance) {
                    // Keep track of points that are the same taxi distance
                        arrTies.Add(i);
                }
            }
            if (iMinDistance > 1) {
                // Not connected, so quit
                break;
            } else {
                // At least one point connected
                if (arrTies.GetCount()>1) {
                        // More than one point that is connected
                        // Choose the one that has only one connection left
                        int iColumnOne = -1;
                        for (int iTie=0; iTie<arrTies.GetCount(); iTie++) {
                                int iAvail = arrTies[iTie];
                                int iPoint = arrAvail[iAvail];
                                int iAttach = 0;
                                for (int i=0; i<arrAvail.GetCount(); i++) {
                                        if (i != iAvail) {
                                                int iCheck = arrAvail[i];
                                                int iDistance =
                                abs(arrPoints[iPoint].x - arrPoints[iCheck].x) +
                                abs(arrPoints[iPoint].y - arrPoints[iCheck].y);
                                                if (iDistance == 1) {
                                                        iAttach++;
                                                }
                                        }
                                }
                                if (iAttach == 0) {
                                        // Doubles back on self, special case
```

```
                                        int iPrev = arrOrder[arrOrder.GetCount()-1];
                                        arrOrder.Add(iPoint);
                                        arrAvail.Add(iPrev);
                                        iClosest = iPoint;
                                        iClosestIndex = iAvail;
                                        break;
                                } else if (iAttach == 1) {
                                        if (iColumnOne <= arrPoints[iPoint].x) {
                                                iColumnOne = arrPoints[iPoint].x;
                                                iClosest = iPoint;
                                                iClosestIndex = iAvail;
                                        }
                                }
                        }
                }
                iCurrent = iClosest;
                arrOrder.Add(iCurrent);
                arrAvail.RemoveAt(iClosestIndex);
                pntCurrent = arrPoints[iCurrent];
            }
        }
        return TRUE;
    }
    return FALSE;
}
```

Pacific Northwest
NATIONAL LABORATORY

U.S. DEPARTMENT OF
ENERGY

Proudly Operated by Battelle *Since 1965*

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnl.gov