Norsk Regnesentral
Norwegian Computing Center
(+47) 22 85 25 00
nr@nr.no · www.nr.no

Note

Functions added to NRLib

During internship on project SeismicForward

Note no SAND/11/11
Authors Marie Lilleborge
Date July 2011

Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

About the authors

Marie Lilleborge did a summer internship at NR in 2011.

Norsk Regnesentral

Norsk Regnesentral (Norwegian Computing Center, NR) is a private, independent, non-profit foundation established in 1952. NR carries out contract research and development projects in the areas of information and communication technology and applied statistical modelling. The clients are a broad range of industrial, commercial and public service organizations in the national as well as the international market. Our scientific and technical capabilities are further developed in co-operation with The Research Council of Norway and key customers. The results of our projects may take the form of reports, software, prototypes, and short courses. A proof of the confidence and appreciation our clients have for us is given by the fact that most of our new contracts are signed with previous customers.

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
N-0314 Oslo, Norway

Besøksadresse
office address
Gaustadalléen 23A, B
N-0371 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Bankkonto · bank account
8200.01.48888
Foretaksnr. · enterprise no.
NO 952125001 VAT

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Title	**Functions added to NRLib**
Authors	**Marie Lilleborge**
Quality assurance	<Insert quality assurance responsible here>
Date	July 2011
Year	2011
Publication number	SAND/11/11

Abstract

This is an overview and description of the C++ code I added to NRLib during my summer internship at NR in 2011.

Keywords	
Target group	SAND group
Availability	Open
Project number	676006
Research field	Seismic data
Number of pages	16

© Copyright Norsk Regnesentral

1 Finding the convex hull of a set of points

Name: GetConvexHull
Member of class: PointSetSurface
Private functions added: AngleSort, RemoveEqualAngles
Input: Called from a PointSetSurface, which is a set of points with x, y and z-coordinates. The GetConvexHull function treats the object as a 2D object, as only the x and y-coordinates are used.
Returns: A polygon variable which contains the convex hull of the PointSetSurface the function is called from. The Polygon points are listed counter clockwise, with the point with the lowest y-coordinate first.

The algorithm implemented is the Graham's Scan. Pseudo code for the Graham's Scan algorithm can be found in Thomas H. Cormen, "Introduction to Algorithms" (2nd printing, 1990). The algorithm can also be found at http://en.wikipedia.org/wiki/Graham_scan. The convex hull of a set of points is the convex polygon that contains all points in the set, either on the inside or along its boundary. Convex means that given any two polygon points, the line segment between them are contained inside the polygon or along its boundary. The Graham's Scan uses the fact that the convex hull of a set of points always contains the point with the smallest y-coordinate, and the points with the smallest and greatest polar angle with respect to this point

The GetConvexHull function first finds the point with the smallest y-coordinate, call this point p0. Then it uses AngleSort to sort the rest of the points according to the polar angle with respect to p0, and then RemoveEqualAngles is called to remove the points that lie on a line segment between p0 and another point. Recursively, in the nth loop, GetConvexHull finds the convex hull of the first n points in the sorted list, together with p0. For each point added to the candidate for the hull, it checks on whether the previous point, or points, needs to be deleted.

AngleSort is a kind of merge sort, which uses the sign of cross products to compare two points. Cross product is also the tool used in RemoveEqualAngles. Both algorithm neglects the z-coordinate of the points, since GetConvexHull looks for the convex hull of the points in the xy-plane.

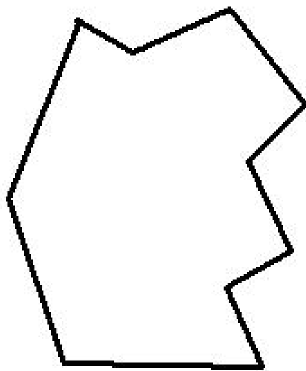
2 Check whether a polygon is convex, and a function to make it convex if not

Name: IsConvex
Member of class: Polygon
Functions called by IsConvex: IsCounterClockwise
Input: Called from a Polygon variable, which is a set of points with x, y and z-coordinates. The IsConvex function treats the object as a 2D object, as only the x and y-coordinates are used. The points are assumed to construct a simple polygon.
Returns: A Boolean variable indicating whether the input represents a convex polygon or not.

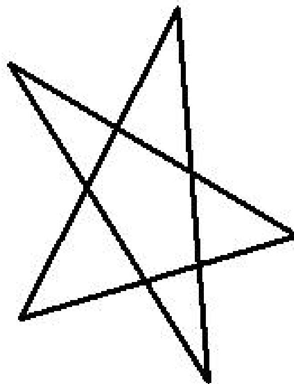
Name: MakeConvex
Member of class: Polygon
Functions called by MakeConvex: IsConvex, IsCounterClockwise
Input: Called from a Polygon variable, which is a set of points with x, y and z-coordinates. The MakeConvex function treats the object as a 2D object, as only the x and y-coordinates are used. The points are assumed to construct a simple polygon.
Returns: Points are deleted from the input Polygon until it is convex.

A simple polygon is a polygon that is not self-intersecting.

Simple polygon

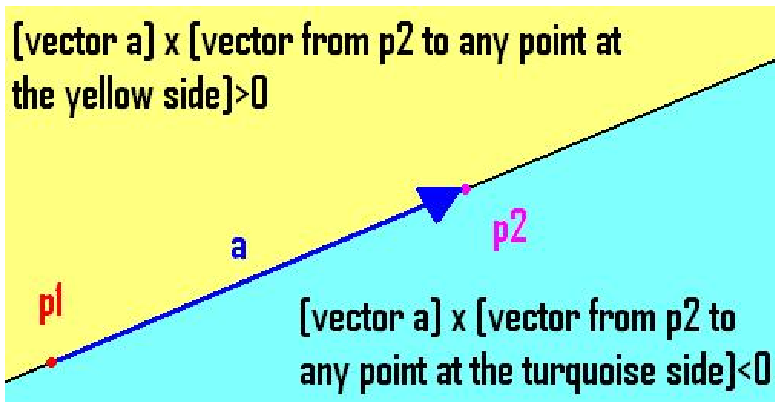


Not a simple polygon



IsCounterClockwise loops over the Polygon points $\mathbf{p}_i = (\mathbf{x}_i, \mathbf{y}_i)$ and calculates the quantity $\sum_{i=0}^{n-2} x_i y_{i+1} - x_{i+1} y_i$, which is positive if the polygon points are ordered counter clockwise and negative if the Polygon points are ordered clockwise.

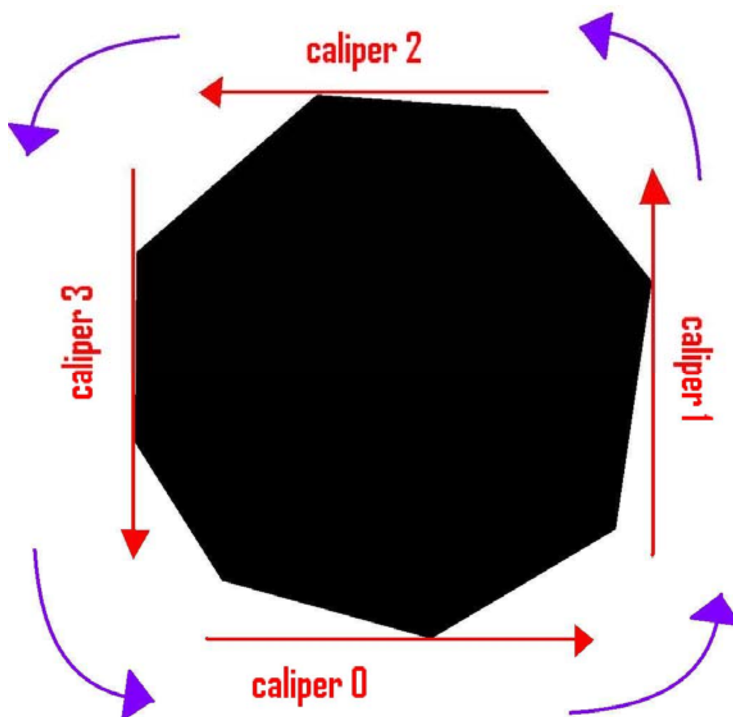
IsConvex loops the Polygon points, and uses the sign of cross products between neighboring polygon corner vectors to check whether the Polygon is convex (if convex, all cross products will have the same sign).



MakeConvex is just an expansion of IsConvex, which deletes a point that makes a not convex corner as long as the polygon is not convex. Note that this function is not designed to find the convex hull of a point set, neither to make an arbitrary polygon convex. The GetConvexHull algorithm in the PointSetSurface class is probably more efficient if more points need to be deleted, and it has no assumptions on the order in which the points are listed. Also note that the convex hull is a unique feature of a set of points, independent of the order. However, MakeConvex might end up with different Polygons for different point orderings, since it assumes that the input is a simple polygon.

3 Minimum area enclosing rectangle for a polygon

Name: MinEnclosingRectangle
Member of class: Polygon
Input: Called from a Polygon variable, which is a set of points with x, y and z-coordinates. The IsConvex function treats the object as a 2D object, as only the x and y-coordinates are used. The points are assumed to construct a simple polygon.
Returns: The variables *x0*, *y0*, *length1*, *length2* and *angle* are called by reference. After the function call, these variables are set to values that encode the minimum area enclosing rectangle for the input Polygon. The *angle* parameter is the angle, in radians, the rectangle is rotated with respect to the x-axis. This is always a number between zero and pi half. The *x0* and *y0* parameters are the coordinates of what would be the corner of minimum x-value and minimum y-value if the rectangle was rotated *angle* radians backwards. In this position, aligned with the x and y-axis, *length1* would be the length of the rectangle along the x-axis and *length2* would be the length of the rectangle along the y-axis.



The algorithm implemented uses the idea of Rotating Calipers, as described in http://cgm.cs.mcgill.ca/~orm/maer.html. The main idea of the algorithm is that the wanted rectangle always has a side that is parallel to one of the polygon sides. Therefore, the algorithm uses four unit vectors in the xy-plane, listed counter clockwise, and each normal to the previous one. These vectors are the so-called calipers. In each step, one of them is parallel to one of the sides of the polygon, and the extreme points of the polygon in the direction of each of the calipers are calculated. These extreme points then describes the minimum area enclosing rectangle aligned with the calipers, and the values corresponding to the currently found rectangle of smallest area independent of angle are kept. After a rotation of pi half radians, all candidate rectangles are checked, and we are done.

Finding the intercept between two pillars (equations used at the end)
Want the (min_x, min_y) corner, that is, the intercept between the extensions of "caliper 0" and "caliper 3". Let the coordinates of that corner be **(x,y)**, the coordinates of the point at which caliper i lies be **(x_i, y_i)**, and denote caliper i by **c_i=[c_{ix},c_{iy}]**.
We know $x = x_3 + t * c_{3x} = x_0 + s * c_{0x}$ and $y = y_3 + t * c_{3y} = y_0 + s * c_{0y}$ for some s and t.
Defining **$\Delta x = x_3 - x_0$** and **$\Delta y = y_3 - y_0$**, we get the linear system

$$\begin{matrix} \Delta x \\ \Delta y \end{matrix} = \begin{bmatrix} c_{0x} & -c_{3x} \\ c_{0y} & -c_{3y} \end{bmatrix} * \begin{matrix} s \\ t \end{matrix}$$

Since the determinant is

$$c_{3x}c_{0y} - c_{3y}c_{0x} = \cos(\propto)\sin\left(\propto + \frac{\pi}{2}\right) - \sin(\propto)\cos\left(\propto + \frac{\pi}{2}\right) = \cos^2 \propto + \sin^2 \propto = 1,$$

the solution of the linear system is

$$\begin{matrix} s \\ t \end{matrix} = \begin{bmatrix} -c_{3y} & c_{3x} \\ -c_{0y} & c_{0x} \end{bmatrix} * \begin{matrix} \Delta x \\ \Delta y \end{matrix}$$

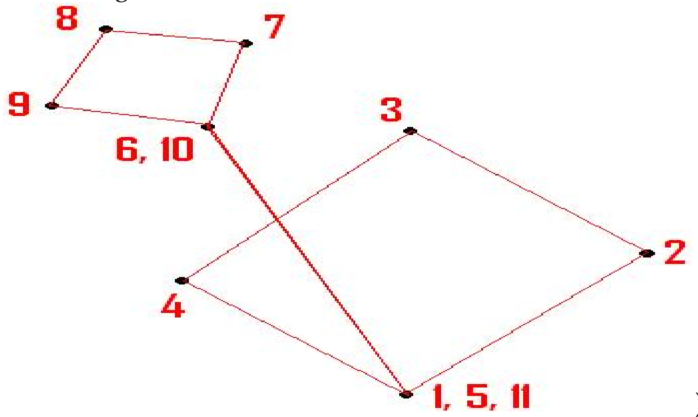
Thus, we can write

$$\begin{matrix} x \\ y \end{matrix} = \begin{matrix} x_3 + c_{3x}(c_{0x}\Delta y - c_{0y}\Delta x) \\ y_3 + c_{3y}(c_{0x}\Delta y - c_{0y}\Delta x) \end{matrix}$$

4 Find polygons around given area

Name: `FindPolygonAroundActivePillars`
Member of class: EclipseGeometry
Private functions added: SearchUp, SearchDown, SearchRight, SearchLeft
Input: Called from an EclipseGeometry variable, which contains an eclipse grid. It also takes a z-value at which the polygons are to be constructed.
Returns: A Polygon at height determined by the input z-value. The Polygon may be consisting of several nonintersecting simple polygons. The Polygon returned encloses the area of intersection points between active pillars in the eclipse grid and the plane of points with the given z-value. Each of the simple polygons is listed with the same point at the beginning as at the end. If more than one simple polygon, the one with the lowest y-coordinate is listed first, and between each new simple polygon this first polygon's first point is added.

(To illustrate why this way it encloses only the necessary area, think of a draw-between-the-numbers game.



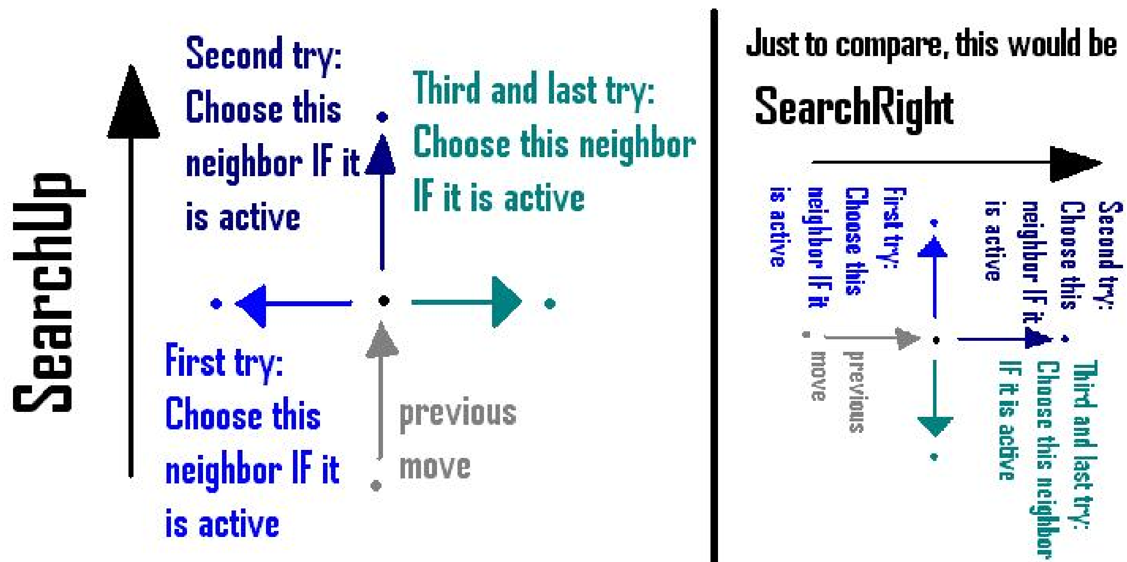
)

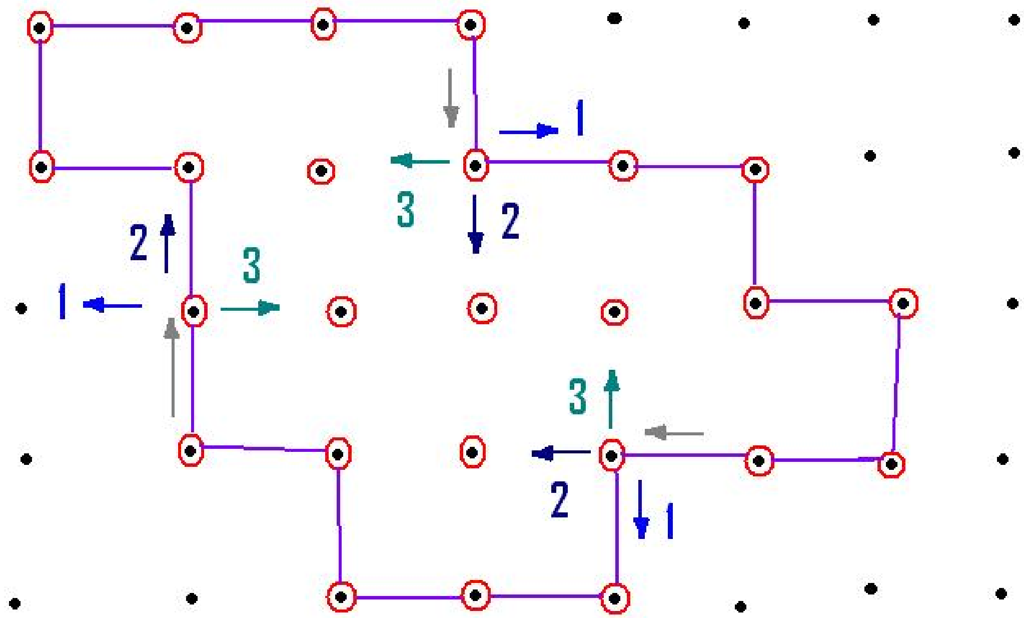
Each eclipse grid has pillars; lines that cross the xy-plane once, and a list of z-values defining the corners of the cells. Each pillar correspond to an index pair (i, j), and together with the pillars corresponding to (i+1,j), (i+1,j+1) and (i,j+1) it determines a column of cells. Each cell is active or not, and a pillar is active if it's collinear with a side of some active cell. Considering two pillars as neighbors if their index pair, **(i_1, j_1)** and **(i_2, j_2),** have $|i_1 - i_2| + |j_1 - j_2| = 1$, we see that the active pillars surround columns that contains some active cell. In the (i, j) grid we can find "islands" of active pillars in the "ocean" of not active pillars. The algorithm finds the simple polygons that encloses each of these islands, and at the end, it merges these into one (not necessarily simple) polygon. Note that the algorithm does not assume any inactive nor any active cells.

The FindPolygonAroundActivePillars algorithm searches through the pillars with a double for loop over the indexes (i, j), looking for active pillars. Each time it finds a point on an until now not discovered border line of an area of active pillars, it starts searching around this border, constructing a simple polygon along the border pillar points with the given z-value. This search around the border is done by the helper functions SearchUp, SearchDown, SearchRight and SearchLeft. Since the original search through the pillars is done by a double for loop over (i, j),

we know that the first active point found has exactly two active neighbors. The one to the left and the one underneath cannot be active, because if so, they would have been found earlier. We also know that any active pillar has at least two active neighbors. The first method called is the SearchUp method (but it could equally well have been the SearchRight method).

The idea behind the search around the border was a recursive search method: As long as you haven't reached where you started, you start the search again in the next point along the border. Given that you name the direction of your last move forward (i.e. went from (i-1, j), now standing in (i, j), forward is in the positive i-direction), you first try your left neighbor, then the one in front, and at last your neighbor to the right. If one of the neighbors is accepted, you don't try the next one(s). This way, since each accepted neighbor is kept in a vector, when we reach our starting point, this vector will contain a simple polygon containing all active pillars in that area. A neighbor is accepted if the corresponding pillar is active, and the search methods are named after what direction is considered as forward at the moment. Up is considered an increase in the j-coordinate, and equivalently increasing i-coordinate is considered right and so on.

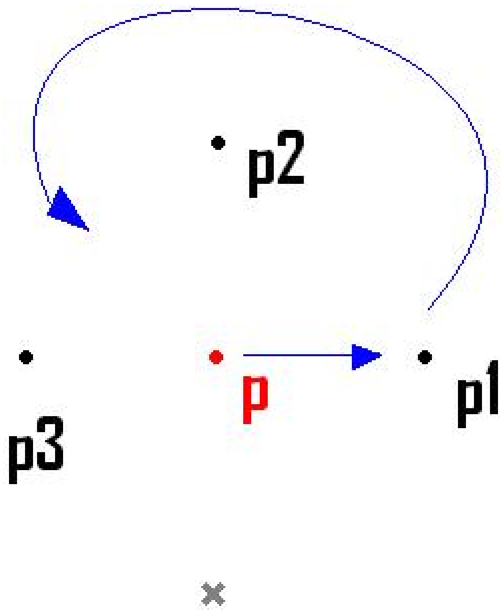



1
3
2
2
3
1
3
2
1

Why this algorithm works

Question: Couldn't we potentially hit a previously visited point that is not the starting point? Answer: First of all, since an active cell (in a column) is equivalent to the four corners (of the column) being active, an active pillar always has at least two active neighbors that is not on opposite sides of that given pillar. So the path around the active pillars will never have to go back and forth along the same line (one active pillar => at least one active cell => four active neighbors around that cell). So we know that the path we want is around a (nonzero) area. The potential problem is if our way of choosing the next cell to search from could make us go to the wrong neighbor. The only way this could happen is if we potentially could hit a counterclockwise loop. **Proof by contradiction**:

Assume p is the first point to be visited twice. This is either the starting point (and then the neighbor underneath is inactive) or WLOG (by symmetry) the first time we visited p was by the SearchUp method. In both cases, the first time we visited p, the undiscovered neighbors are the one to the left, say p3, the one over, p2, and the one to the right, p1. To end up in a counterclockwise loop, we could either go to p1 and then revisit p from p3 or p2 the second time, or go to p2 and arrive from p3 the second time. But none of these cases would ever occur: The first case implies both p1 and either p2 or p3 active, which in turn implies that we would not go to p1 at the first move from p. The second case implies p2 and p3 active, which in turn implies that we would move left from p the first time.

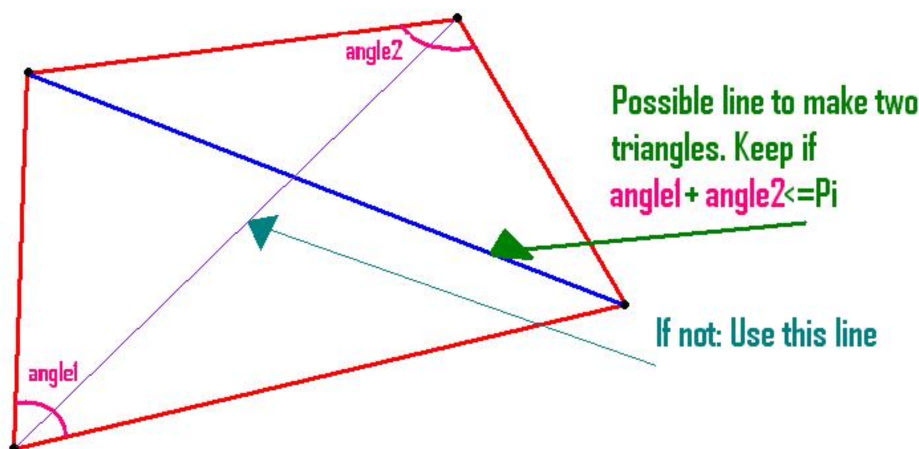


5 Find surface heights inside a rectangle for a given layer

Name: FindLayerSurface
Member of class: EclipseGeometry
Private functions added: TriangularFillInZValuesInArea, BilinearFillInZValuesInArea, FillInZVAluesByAveraging
Input: Called from an EclipseGeometry variable, which contains an eclipse grid. Three doubles indicating xy-coordinates of one corner of the rectangle we want surface heights for, and its polar angle. Other input: Layer number k, an integer indicating top or bottom layer (following previous standard), a Boolean variable indicating whether the surface heights inside a cell should be done by triangularization or by the bilinear formula. It also takes a Grid2d of doubles to fill the surface heights into. Together with doubles indicating step lengths along the sides of the rectangle, the size of the Grid2d gives the lengths of the sides of the rectangle.
Returns: The values in the Grid2d z_surface are being filled in by the surface height of the given rectangle. Each entry (i, j) now contains the midpoint height of the (i, j)th cell constructed by dividing the given rectangle into equal smaller rectangles of width dx and length dy.

FindLayerSurface loops all eclipse cells, and for each eclipse cell it fills out all entries in z_surface that corresponds to a midpoint in that eclipse cell. The writing in z_surface is done by TriangularFillInZValues or BilinearFillInZValues (which method that is to be used is chosen by an input Boolean variable). While looping the eclipse grid the method also checks whether there's a fault along the i- and j-coordinates, respectively, by checking whether the neighboring cells have the corresponding corners in common (whether they have the same x- and y-coordinates). If a fault is detected between two cells, the gap between them is filled as if it was an eclipse cell with heights from the (just mentioned) neighboring (real) eclipse cells.

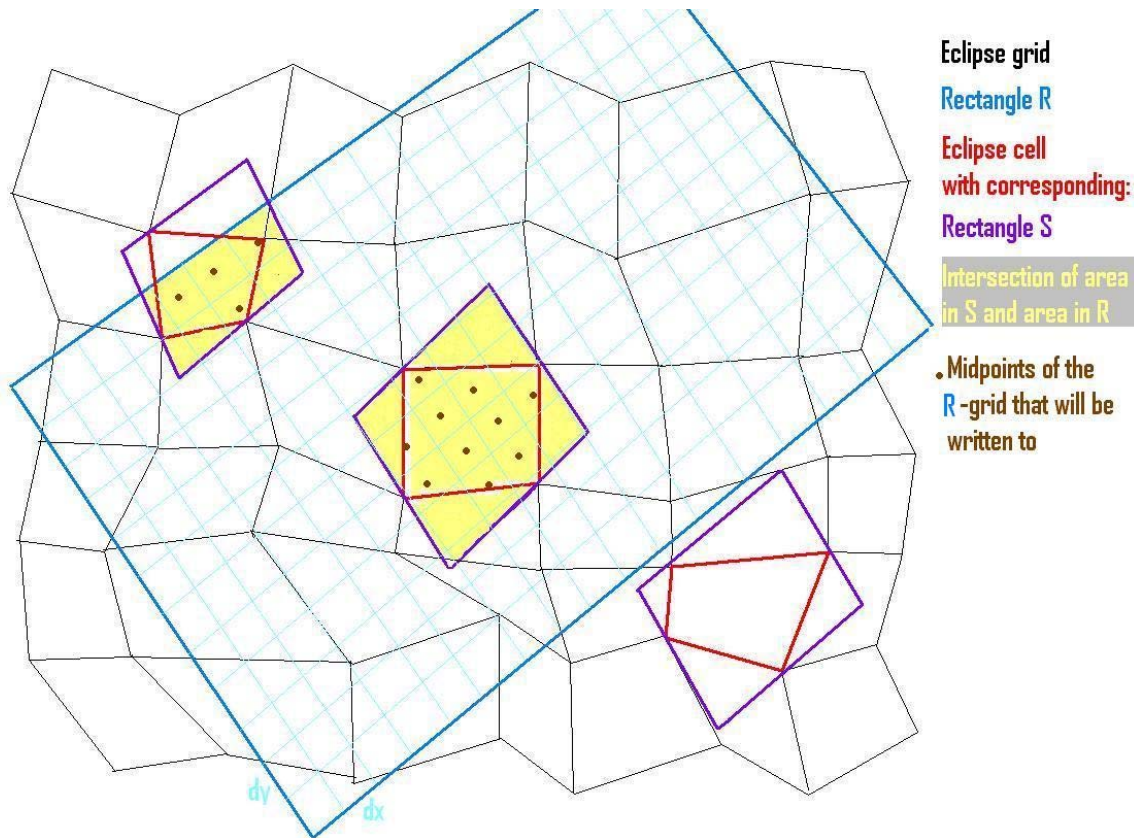
TriangularFillInZValues divides the four-sided eclipse cell into two triangles by Delaunay decomposition in the xy-plane:



Inside each triangle it fills out 'the height of the plane determined by the corner points' in each (i, j) cell of z_surface that corresponds to xy-coordinates inside the triangle. This is done by member functions of the Triangle class.

BilinearFillInZValues calculates height of midpoints using to member functions of the BilinearSurface class.

In common for BilinearFillInZValues and TriangularFillInZValues:
We are given a polygon described by its four corner points, and a rectangle R (in some sense equivalent to a grid we want to write values to). To loop over the entries of the R-grid more easily, a linear transformation (i.e. rotation) of the coordinates of the polygon corners are done s.t. the coordinate axis are parallel to the rectangle sides. From these coordinates we find the extreme x- and y- values of the polygon (i.e. a rectangle S which contains the polygon that also has the same polar angle as R). Then we loop over the intersection of S and R to fill in the entries that lies inside the polygon we started with.



FillInZValuesByAveraging starts in the “center of mass” of the entries in z_surface that has written a value to it. Then it works outwards in squares of increasing “radii”: For each entry that has not got a value written to it yet, the average of the values written to its neighbors is filled in. This routine is done in case R has some nonempty intersection with the compliment of the eclipse grid for the given layer. If this occurs, we set the values outside the eclipse grid by using this method. If the intersection is empty or all of R, the algorithm will not write any values to z_surface.

6 Other changes I made to the Visual Studio project

In some of the .h or .cpp files I had to include some NRLib .h or .cpp files, these **#include**s are all marked with //M at the end (probably no need for it, but just in case).

I also edited the SeismicForward member function FindVpAndR.
Changes:
1. Delaunay Decomposition to make triangles (no longer always a triangle consisting of pt1, pt2 and pt4)
2. Limiting which entries in vpgrid we need to check if is inside the eclipse cell (same idea as in the FindLayerSurface routine.
3. Filling out values corresponding to "frame" (inside eclipse grid, but outside midpoints of the outermost eclipse grid cells). These values are filled out by mirroring out values to the outside of the eclipse grid, and then filling out unwritten values on the inside by triangulating: