Sameer Pusegaonkar
CMSC498Q Project Report

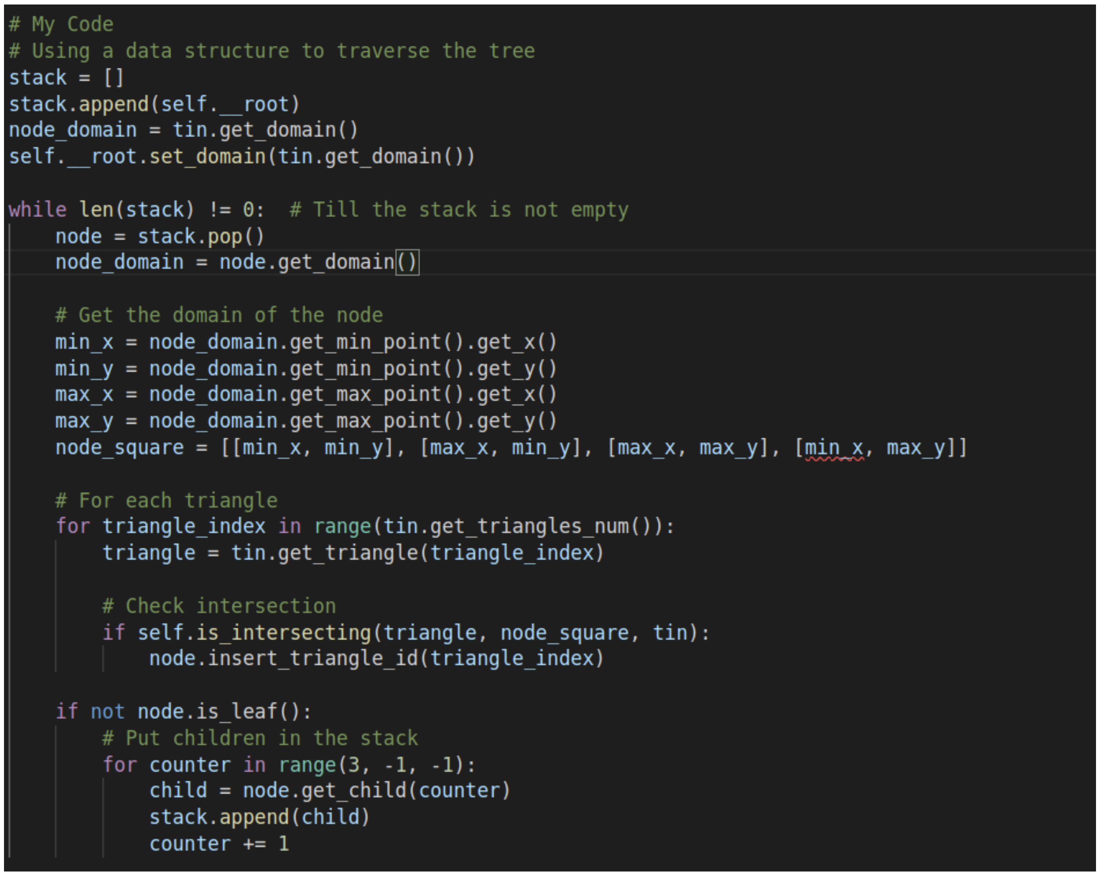
---

1. **Build a Terrain tree, which considers the intersection between triangles and blocks.**

The idea behind building a Terrain tree is to get all intersecting triangles in particular node.

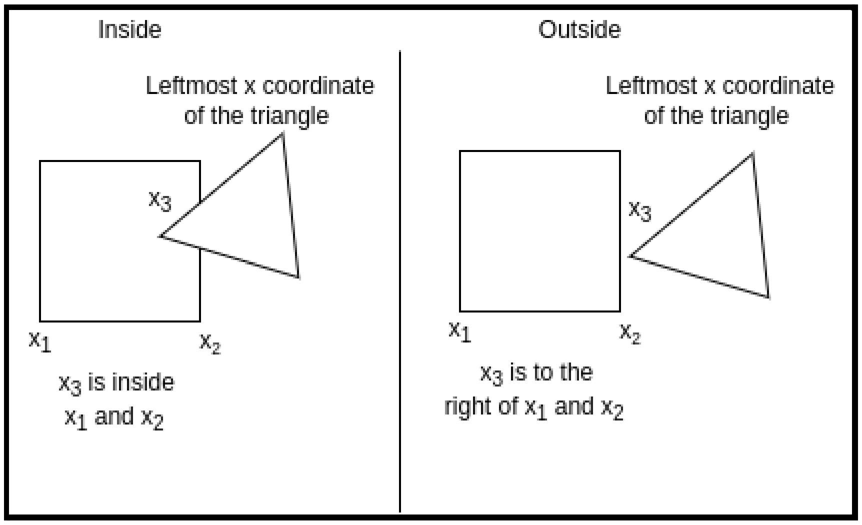
Process: First we build the tree by inserting each and every vertex into the tree using the insert_vertex() method. Next, we perform a level order traversal. Note that the order of traversal does not matter. This process can be done using inorder, postorder or preorder traversal as well.

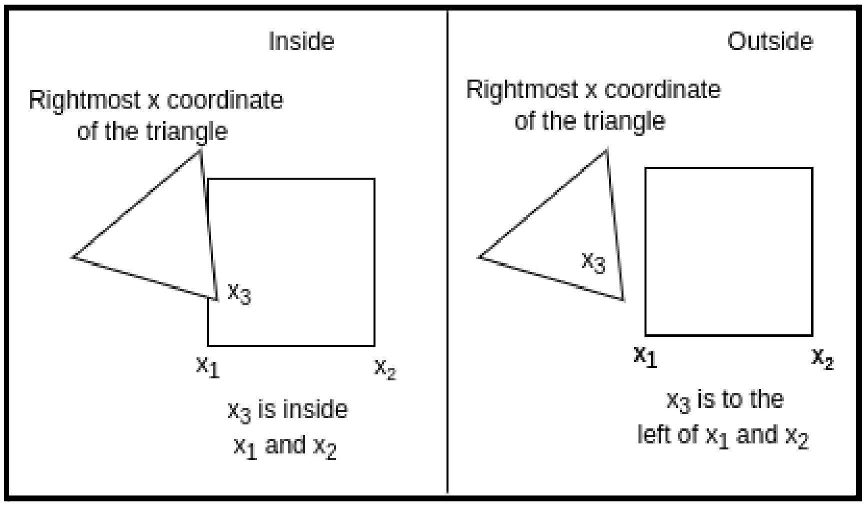
This helps us to retrieve each and every node. We check if the node is intersecting with all the triangles by using cross products.



```python
# My Code
# Using a data structure to traverse the tree
stack = []
stack.append(self.__root)
node_domain = tin.get_domain()
self.__root.set_domain(tin.get_domain())

while len(stack) != 0:  # Till the stack is not empty
    node = stack.pop()
    node_domain = node.get_domain()

    # Get the domain of the node
    min_x = node_domain.get_min_point().get_x()
    min_y = node_domain.get_min_point().get_y()
    max_x = node_domain.get_max_point().get_x()
    max_y = node_domain.get_max_point().get_y()
    node_square = [[min_x, min_y], [max_x, min_y], [max_x, max_y], [min_x, max_y]]

    # For each triangle
    for triangle_index in range(tin.get_triangles_num()):
        triangle = tin.get_triangle(triangle_index)

        # Check intersection
        if self.is_intersecting(triangle, node_square, tin):
            node.insert_triangle_id(triangle_index)

    if not node.is_leaf():
        # Put children in the stack
        for counter in range(3, -1, -1):
            child = node.get_child(counter)
            stack.append(child)
            counter += 1
```

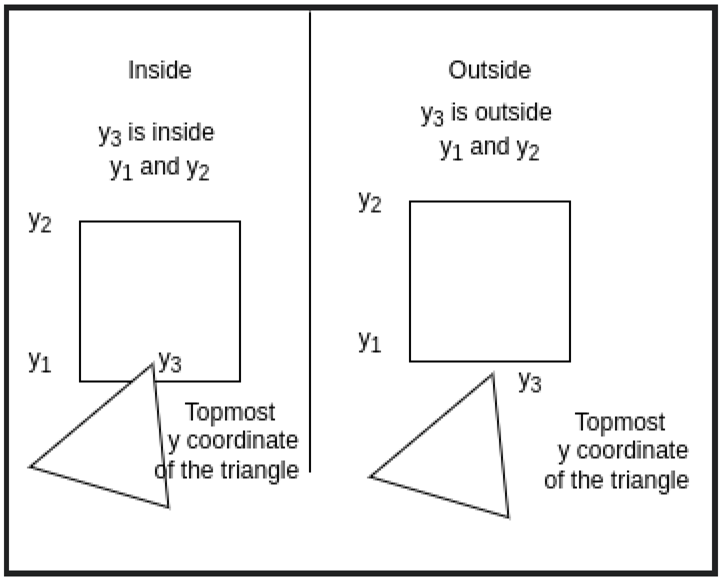
First we check the **boundary conditions**. We check if the triangle t's leftmost x coordinate lies in min and max x positions of the node domain . Similarly, we check if the triangle t rightmost x coordinate doesn't lie between the min and max x position of the node domain.

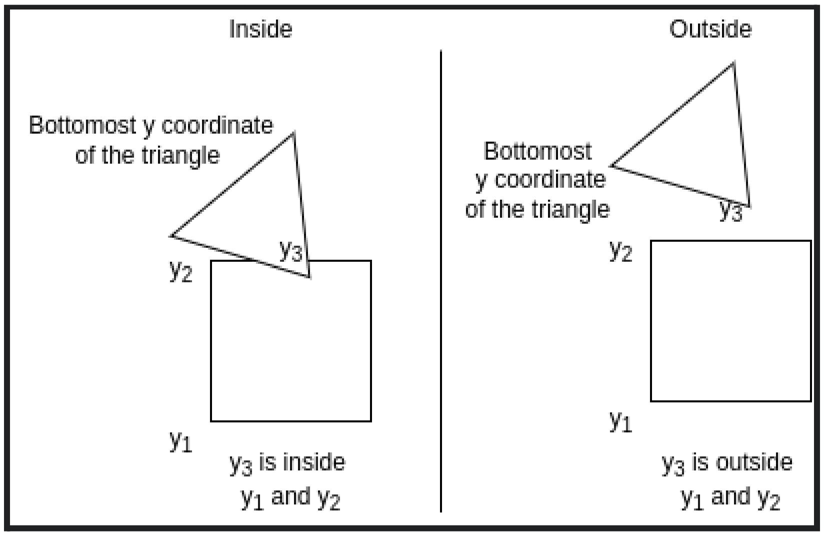

Inside
Outside
Leftmost x coordinate of the triangle
x_3
x_1
x_2
x_3 is inside x_1 and x_2
Leftmost x coordinate of the triangle
x_3
x_1
x_2
x_3 is to the right of x_1 and x_2


Inside
Outside
Rightmost x coordinate of the triangle
x_3
x_1
x_2
x_3 is inside x_1 and x_2
Rightmost x coordinate of the triangle
x_3
x_1
x_2
x_3 is to the left of x_1 and x_2

We do the same operations for y coordinate.

We check if the triangle t’s bottommost y coordinate doesn't lie between the min and max positions of y. Similarly, we check if the triangle TINs topmost y coordinate doesn't lie between the min and max positions of y.


Inside
y_3 is inside y_1 and y_2
y_2
y_1
y_3
Topmost y coordinate of the triangle
Outside
y_3 is outside y_1 and y_2
y_2
y_1
y_3
Topmost y coordinate of the triangle


Inside
Bottomost y coordinate of the triangle
y_3
y_2
y_1
y_3 is inside y_1 and y_2
Outside
Bottomost y coordinate of the triangle
y_3
y_2
y_1
y_3 is outside y_1 and y_2

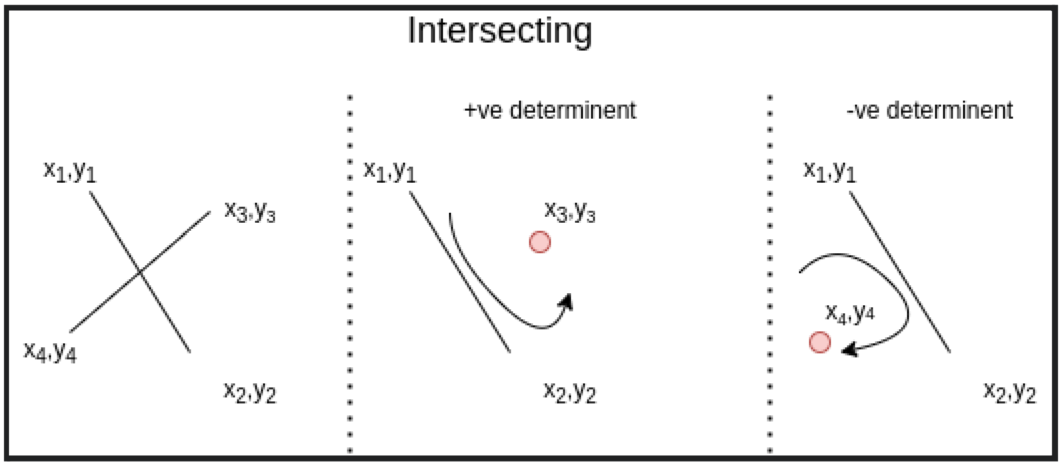
This allows us to quickly determine if the extreme points of the triangles are outside the node domain. Using this technique we do not need to compute further computations if we know the triangle is outside the node domain.

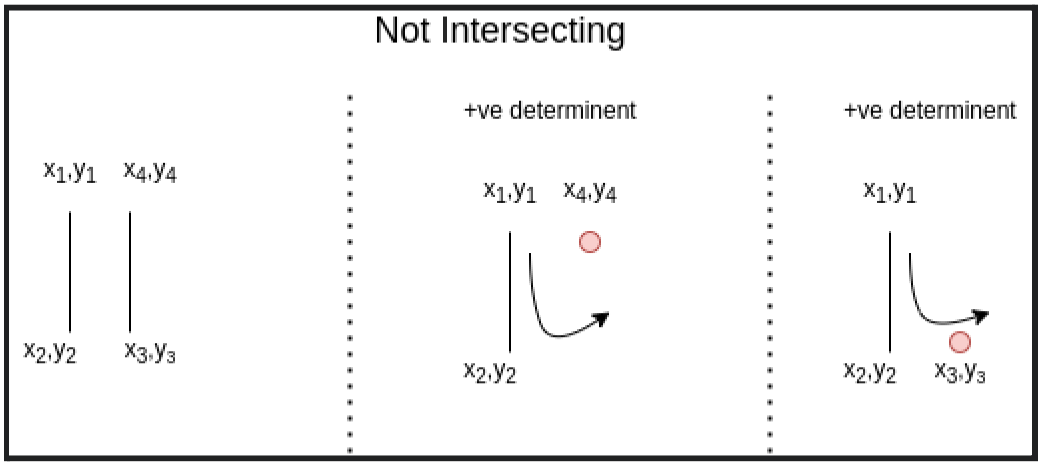
Further, we need to check if the Tin triangle and the square node are intersecting. This is done by checking that **each edge of a triangle intersects with any other edge of the node domain.**

This is a fundamental problem of checking intersection between 2 line segments. This can be done by checking cross products. Since a square has 4 sides and a triangle has 3 sides there will be a total of 12 comparisons

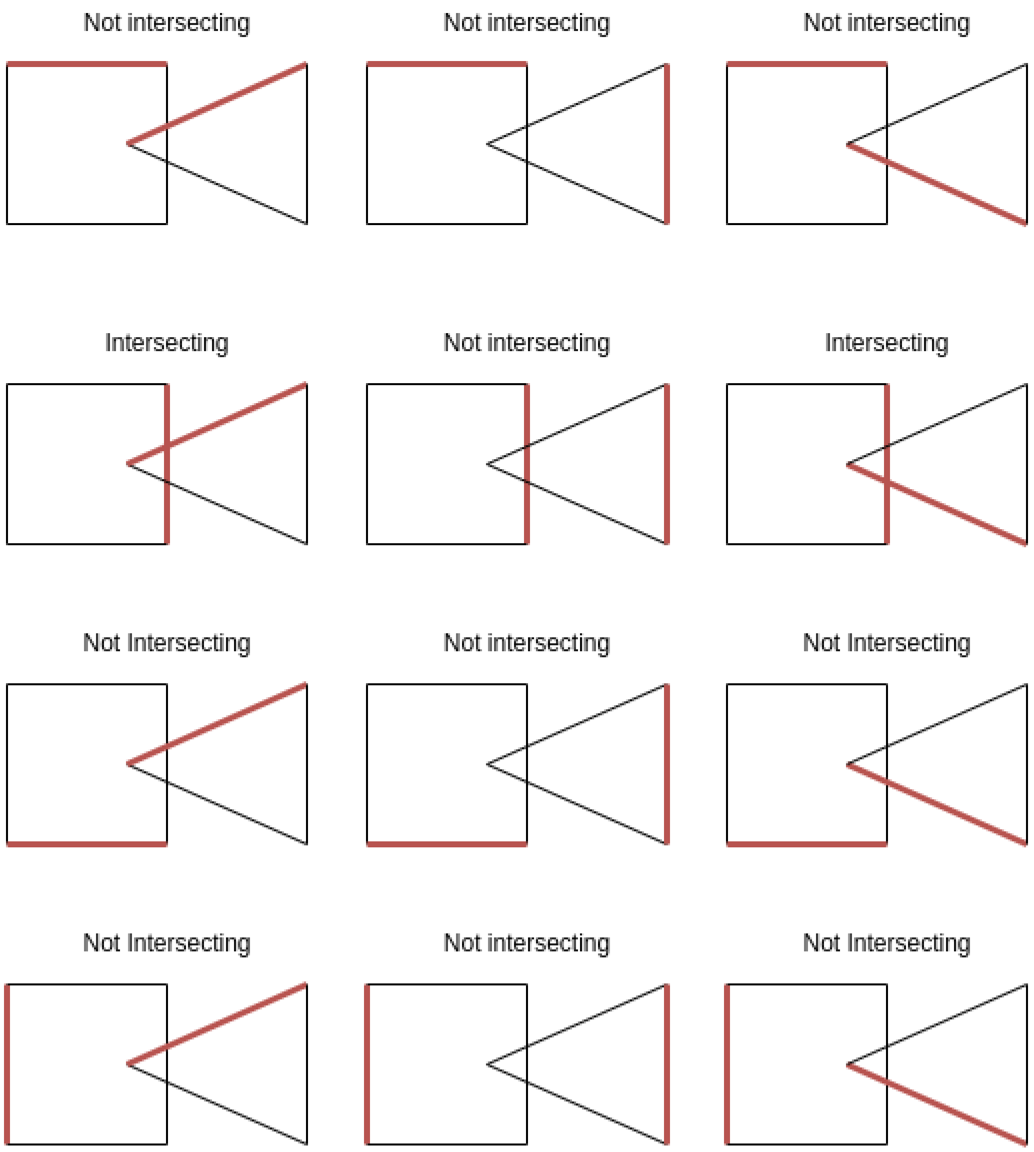
Consider the intersection of 1 edge of the node square and 1 edge of the triangle.

Let (x1,y1 x2, y2)be the edge of the node square and (x3,y3 x4,y4) be the triangle edge. The cross product is calculated wrt to a point in one edge and comparing it with the other edge



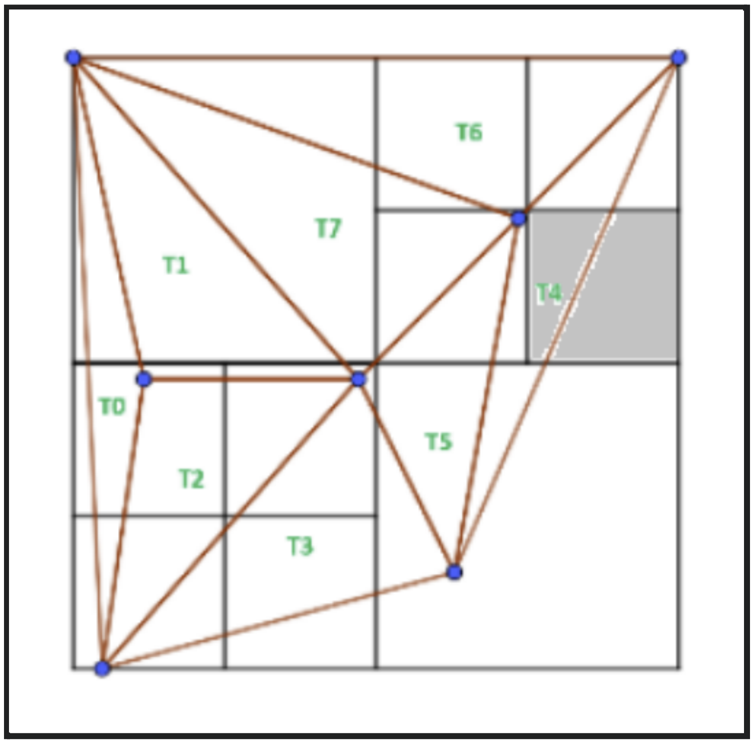


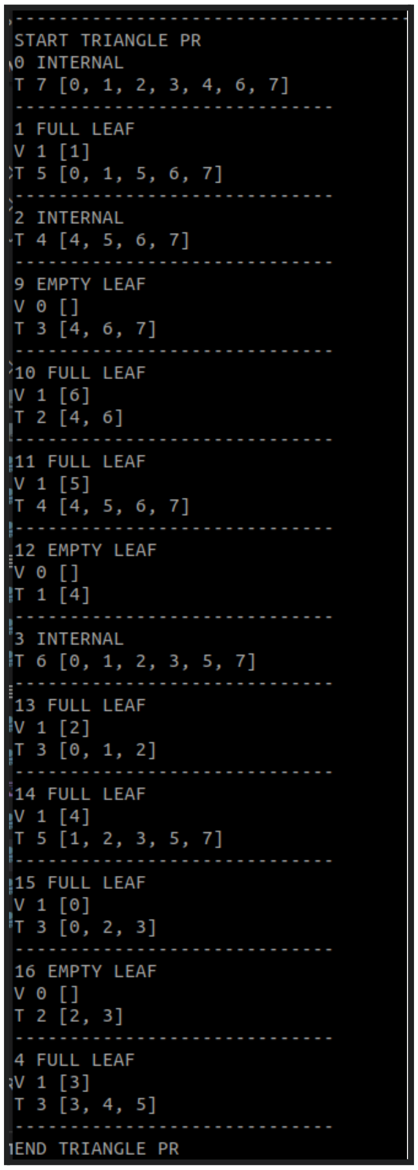
Therefore we need mismatching orientations for the 2 edges to be intersecting



All the iterations of checking intersections. The red lines indicate the edges under consideration for checking intersections. In total we will have 12 comparisions

An example output is given below:





```
---------------------------------------
START TRIANGLE PR
0 INTERNAL
T 7 [0, 1, 2, 3, 4, 6, 7]
-----------------------------
1 FULL LEAF
V 1 [1]
T 5 [0, 1, 5, 6, 7]
-----------------------------
2 INTERNAL
T 4 [4, 5, 6, 7]
-----------------------------
9 EMPTY LEAF
V 0 []
T 3 [4, 6, 7]
-----------------------------
10 FULL LEAF
V 1 [6]
T 2 [4, 6]
-----------------------------
11 FULL LEAF
V 1 [5]
T 4 [4, 5, 6, 7]
-----------------------------
12 EMPTY LEAF
V 0 []
T 1 [4]
-----------------------------
3 INTERNAL
T 6 [0, 1, 2, 3, 5, 7]
-----------------------------
13 FULL LEAF
V 1 [2]
T 3 [0, 1, 2]
-----------------------------
14 FULL LEAF
V 1 [4]
T 5 [1, 2, 3, 5, 7]
-----------------------------
15 FULL LEAF
V 1 [0]
T 3 [0, 2, 3]
-----------------------------
16 EMPTY LEAF
V 0 []
T 2 [2, 3]
-----------------------------
4 FULL LEAF
V 1 [3]
T 3 [3, 4, 5]
-----------------------------
END TRIANGLE PR
```

The format is as follows

N LABEL_definition (N is the label, definition can indicate ful leaf, internal or empty leaf)

V number_of_index vertex_index

T number_of_intersecting_triangles intersecting_triangles_indices

----- indicates a separation between 2 nodes

Since output text for other files is huge, it is not included here. Instructions to run Q3 can be found in the readme file.

Time Complexity: The time complexity of this algorithm in the worst case is O(T*N) where T is the number of triangles and N is the number of nodes.

1.2. Given a grid, implement a query that can extract elevations at grid vertices. Output file format: a matrix of row column size, each value corresponds to the elevation of one vertex.

We first create a grid matrix which is made of the Vertex class.
The origin of the grid is the bottom left corner. Given the starting points of the grid we can compute the x and y coordinate of the entire grid vertices.

The idea is that for each vertex of the grid we find the z value of the point by interpolating using which triangles are intersecting on the vertex.

We first locate the node in which the vertex is located. Finding the node helps us to constrain the search of intersecting triangles.

Since we know which triangle belongs to the node, we now have 3 separate cases:

1) Triangle Vertex is on the the grid vertex
2) Triangle Edge is on the grid vertex
3) Grid vertex is inside the triangle

Consider a TIN T, a grid G and the vertices of the grid denoted by G_{ij} where i and j is the row and col of the TIN.

To go over how the G is built, we need to first discuss all the possibilities of how a triangle t from T can intersect G_{ij}.

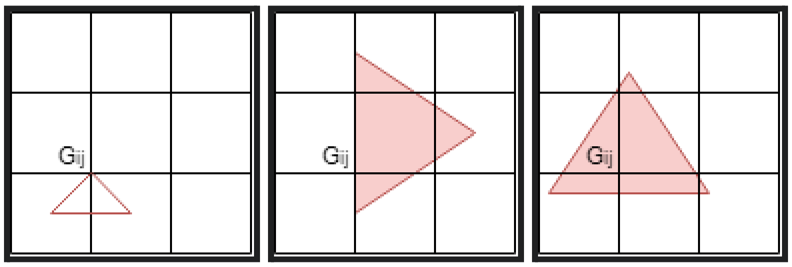


Figure: Case 1, Case 2 and Case 3 from left to right

This process is continued for each grid vertex.

For the implementation, the vertex class is modified to take the z value as optional for now. "Nan" is replaced by None. An object of the grid class is made of n_rows+1 and n_cols+1 Vertex class objects.

Algorithm:

vector<vector<int> getElevationValues(Tin T, start_x, start_y, rows, cols, cell_height, cell_width**):**

```
Step 1) Initialize Grid G of size [m][n]

// The start_x and start_y are at the bottom left corner

Step 2.a) For i in rows+1 to 0:    //decreasing order of i
Step 2.b)       For j in cols+1:      //ascending order of i
Step 2.c)               $G_{ij}$.x = start_x + i * cell_height
Step 2.c)               $G_{ij}$.y = start_y + j * cell_width


Step 2.a) For i in G.length:    // rows
Step 2.b)       For j in G[0].length:    // cols
Step 2.c)               For t in T:    //  For all triangles in TIN
Step 2.d)                       if (i,j) is_a_vertex_of_triangle(t)
Step 2.d.1)                             $G_{ij}$.z = elevation of vertex
Step 2.d.2)                             break
Step 2.e)                       if (i,j) is_on_edge_of_triangle(t)
Step 2.e.1)                             $G_{ij}$.z = interpolateUsing2Points(vertex1,vertex2, (i,j))
Step 2.e.2)                             break
Step 2.f)                       if (i,j) is_inside_triangle(t)
Step 2.f.1)                             $G_{ij}$.z = interpolateUsing3Points(t, (i,j))
Step 2.f.2)                             Break
Step 3) Return G
```

Case 1) We first simply check if the vertex point lies on the triangle vertex if they share x & y coordinates. The z value for this condition can be found by simply assigning the z value of the intersected triangle vertex.

```
Algorithm: is_a_vertex_of_triangle
#The function is called by the grid using i and j coordinates
bool is_a_vertex_of_triangle(triangle t):
Step 1.a) For v in t:
              If v == (i,j):
                  Return True
Step 1.b) return False
```

Case 2) We check if a grid vertex lies on the edge by checking if point orientation wrt the edges is 0. This means that the determinant of the point wrt the edge is 0. This also means that once we obtain a 0 determinant, we need to check if the point lies within the segment bounds.

```
Algorithm: is_on_edge_of_triangle
#The function is called by the grid using i and j coordinates
bool is_on_edge_of_triangle(triangle t):

Step 1.a) v1, v2, v3 = getVertex(t)

// For this method, we will check if the orientation of the point is 0 wrt the vertex pair.
// Cross product is 0 -> point is collinear

Step 2.a) if orientation(v1, v2) == 0 and if ( v1.x <= i <= v2.x and v1.y <= j <= v2.y ):
              Return True
Step 2.b) if orientation(v2, v3) == 0 and if ( v2.x <= i <= v3.x and v2.y <= j <= v3.y ):
              Return True
Step 2.c) if orientation(v3, v1) == 0 and if ( v3.x <= i <= v1.x and v3.y <= j <= v1.y ):
              Return True
Step 2.d) return False
```

The getOrientation() method will simply compute the cross product based on the determinant.

```
Algorithm:
bool getOrientation(Point p1, Point p2,  Point p3):
Step 1)   p3p1X = p1.x - p3.x
Step 2)   p3p1Y = p1.y - p3.y
Step 3)   p3p2X = p2.x - p3.x
Step 4)   p3p2Y = p2.y - p3.y
Step 5)   determinant = (p3p1X * p3p2Y ) - (p3p1Y *p3p2X )
Step 6)   if (determinant > 0):
              return 1
Step 7)   if (determinant < 0):
              return -1
Step 8)   return 0
```

The z value for this condition can be found by computing the dot product. We first project the grid_vertex point on top of the edge. This allows us to see (how far) the vertex is. We then simply traverse from the starting point of the edge towards the grid_vertex point.

```
Algorithm: Interpolation using 2 Points
#The function is called by the grid using i and j coordinates
float interpolateUsing2Points(vertex_1, vertex_2):
Step 1.a) V = vertex_2 - vertex_1

# For this method, we get the orthogonal projection of (i,j) on to the vector vertex_1 to
vertex_2

Step 1.a) d = euclidianDistance(vertex_1, (i,j))
Step 1.b) return vertex_1['z'] + (d/norm(v))* vertex_2['z']
```

Case 3) We now check if a grid vertex is inside the triangle by checking the cross products.

The basic geometric operation is to check the orientation(cross product) of the vertices of the triangle and the point P. If the orientation of P is the same for all vectors in the triangle, the point P is inside, otherwise outside.

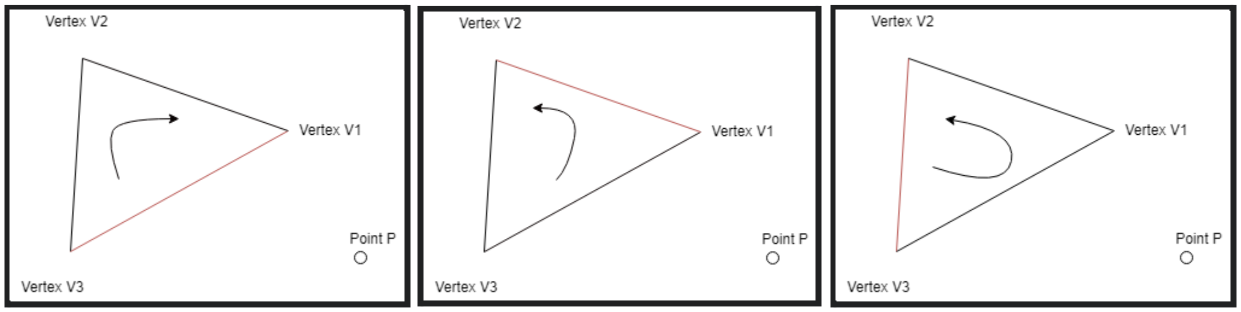


Figure: Considering each vertex pair one by one and checking if P is inside or not. Since all the orientations don't match, P is outside the triangle.

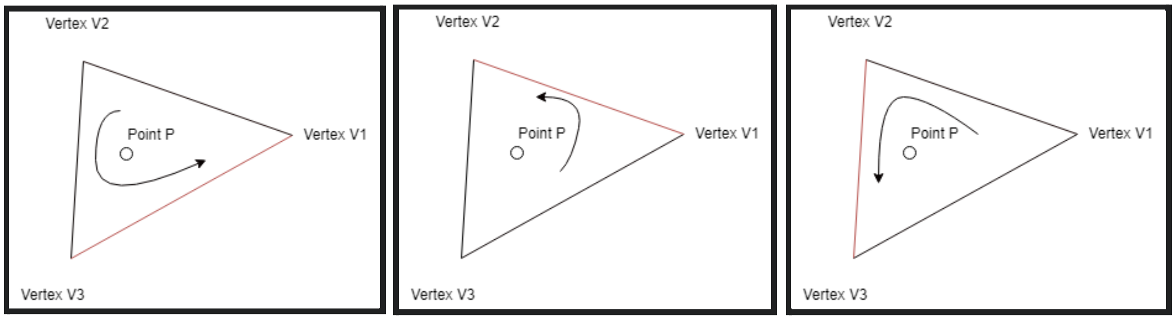


Figure: Considering each vertex pair one by one and checking if P is inside or not. Since all the orientations match, **P is inside the triangle.**

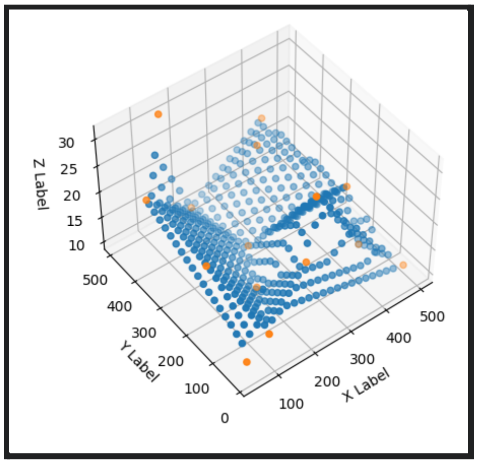
```
Algorithm:
#The function is called by the grid using i and j coordinates
bool is_inside_triangle(triangle t):
Step 1.a) P = (i,j)

# For this method, we will see if the point(i,j) is inside the triangle using a cross product.

Step 1.b) if orientation(v1, v2, P) ==  orientation(v2, v3, P) == orientation(v3, v1, P):
              Return True
Step 1.c) return False
```

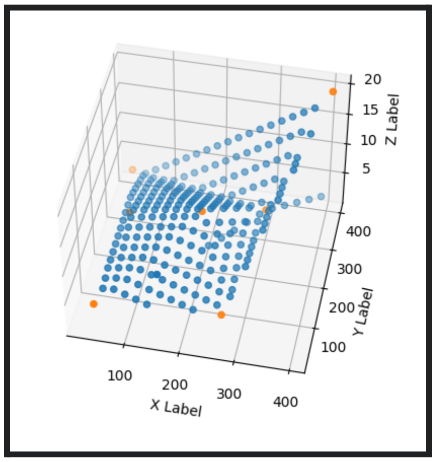
Once determined that the grid_vertex is inside the triangle, we now need to compute the z elevation. We know that the triangle forms a plane. The idea is that the normal of this plane multiplied by a point on the plane will have a dot product of 0. We use this information to deduce.

```
Algorithm: Interpolation using 3 Points (Plane)
#The function is called by the grid using i and j coordinates
float interpolateUsing3Points(triangle t):
Step 1) v1, v2, v3 = getVertices(t)

// For this method, we get z elevation by creating a normal vector which is created by
the plane of v1, v2, v3.
Step 2) N = (p2-p1) x (p3-p1)
```

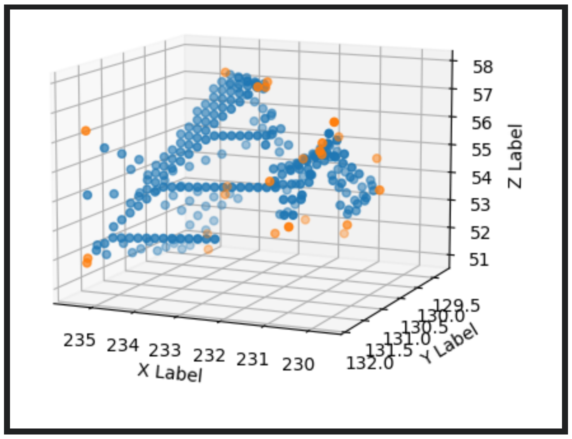
// For point P to be on the same plane, it needs to satisfy the plane equation. (p4-p1).N = 0
// Rearranging we get:
Step 3) z = z1 - ((x4-x1)*N['x'] + (y4-x1)*N['y']) / N['z']
Step 4) return z

Time Complexity: The time complexity in the worst case is O(M*N*logK*T) where T is the number of m = number of row vertices
number of n = number of col vertices
K is the number of nodes in the tree
T is the number of intersecting triangles in the node
for building the grid

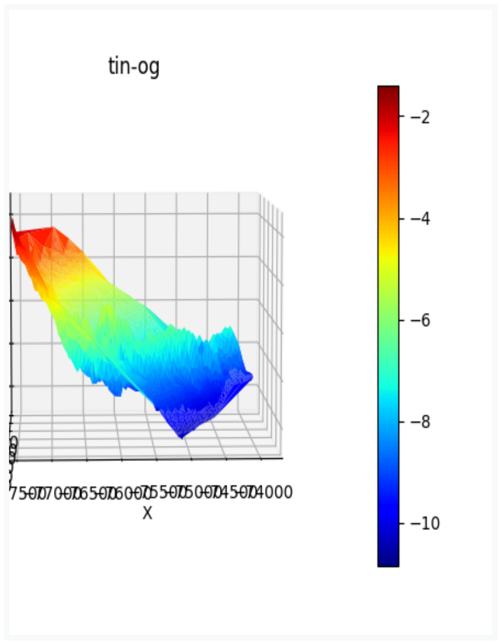


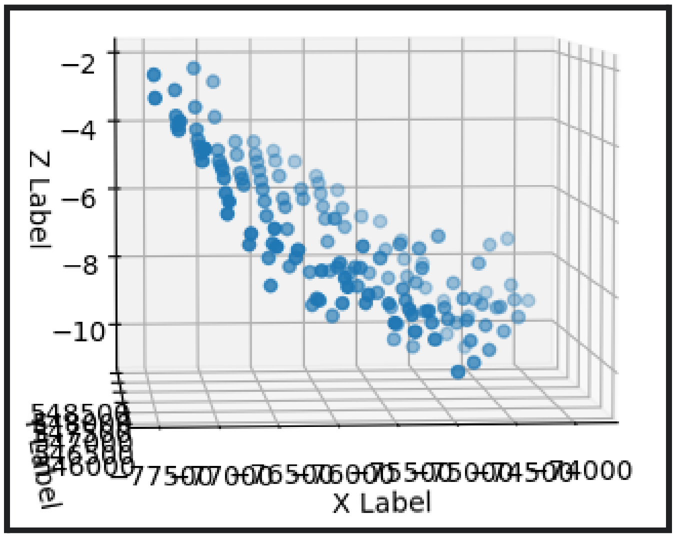


Solution for input_vertices_1, input_vertices_2 file

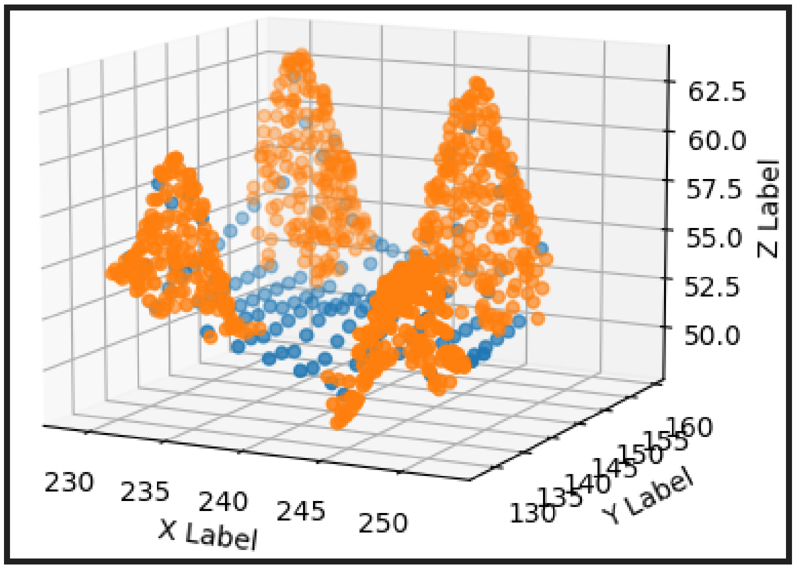


Solution for input_vertices_3 file





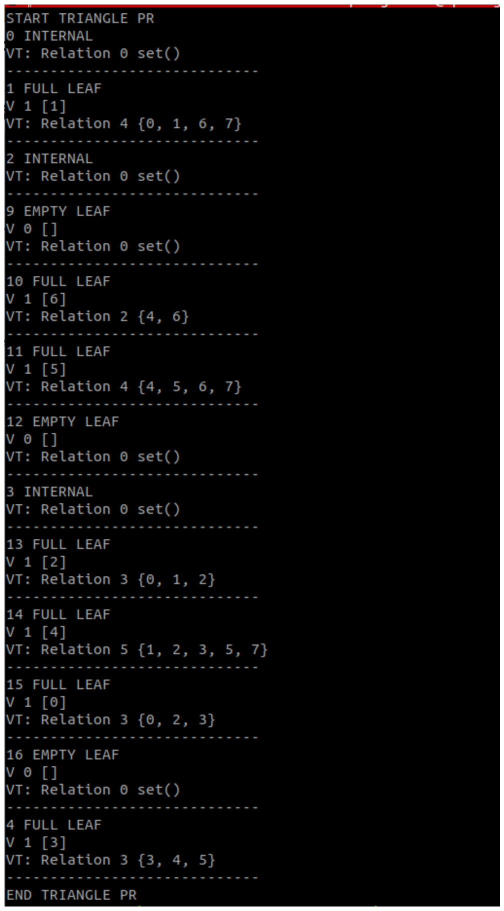
Solution for test_bathymetric.pts .The TIN is shown on the left and the output is shown on the right. Since the point density is large for this file, the output is shown separately on the right.



Solution for test_chm.pts

Q3 Implement a query for extracting the VT relation within a node. Output requirement: you are expected to print out the VT information of vertices within this node when this query is called. You can test it with a pre-order traversal starting from the root.

For each vertex we find the node it belongs to. We check the intersecting triangles from Q1 to see which triangles are intersecting for a particular node. This allows us to search within these constrained triangles. We now check which triangles are incident among these triangles. This is done by checking if the current vertex under query with the triangle vertices. This process is continued for each vertex.



```
START TRIANGLE PR
0 INTERNAL
VT: Relation 0 set()
-----------------------------
1 FULL LEAF
V 1 [1]
VT: Relation 4 {0, 1, 6, 7}
-----------------------------
2 INTERNAL
VT: Relation 0 set()
-----------------------------
9 EMPTY LEAF
V 0 []
VT: Relation 0 set()
-----------------------------
10 FULL LEAF
V 1 [6]
VT: Relation 2 {4, 6}
-----------------------------
11 FULL LEAF
V 1 [5]
VT: Relation 4 {4, 5, 6, 7}
-----------------------------
12 EMPTY LEAF
V 0 []
VT: Relation 0 set()
-----------------------------
3 INTERNAL
VT: Relation 0 set()
-----------------------------
13 FULL LEAF
V 1 [2]
VT: Relation 3 {0, 1, 2}
-----------------------------
14 FULL LEAF
V 1 [4]
VT: Relation 5 {1, 2, 3, 5, 7}
-----------------------------
15 FULL LEAF
V 1 [0]
VT: Relation 3 {0, 2, 3}
-----------------------------
16 EMPTY LEAF
V 0 []
VT: Relation 0 set()
-----------------------------
4 FULL LEAF
V 1 [3]
VT: Relation 3 {3, 4, 5}
-----------------------------
END TRIANGLE PR
```

Output for file input_vertices_1

Since output text for other files is huge, it is not included here. Instructions to run Q3 can be found in the readme file.
Time Complexity: The time complexity for building the entire VT relation in the worst case is O($\log_4$K*T*V)
K is the number of nodes in the tree
T is the number of intersecting triangles in the node
V is the number of vertices

<u>Advantages and Disadvantages of Terrain tree and triangle PR-quadtree are discussed wrt to factors below</u>

<u>**Accessing Triangles:**</u> Accessing all intersecting triangles will be done in O(1) in the Terrain tree. This will have to be computed in a triangle PR quadtree.

<u>Memory Requirements</u>: More memory space is required to store terrain trees as each node will consist of all intersecting triangles. Since only 1 or more troamg;e are stored in a node of a triangle PR quadtree, the node space required for each node will be small**.**

<u>Interpolation:</u> For a particular node, it can help to interpolate a point inside the node if there is an intersecting triangle. Interpolation in a triangle PR quadtree will not be possible as no node will be intersecting.

<u>Querying:</u> Querying can take a complexity of O(T) where T is the number of all triangles for a triangle PR quadtree whereas it will be logarithmic time for a terrain pr-quadtree. Finding incident triangles will also be more efficient in a terrian quad tree as compared to a triangle PR quadtree.