CS478 Computational Geometry

Spring 2021-2022

Implementation of Two Delaunay Triangulation Algorithms and Comparing Their Performance

Final Report

Group Members

Elif Kurtay 21803373

Atakan Dönmez 21803481

Table Of Contents

Project Description

For our project we are implementing two different algorithms for Delaunay Triangulation in two dimensions. Points in a plane form a Delaunay triangulation when the bounded areas formed by all edges are triangles and the circumcircle of the triangles do not contain any other data points inside.

The two algorithms we will use to form Delaunay triangulation out of a data set are the Randomized Incremental Algorithm and the Planar Divide-And-Conquer Algorithm. The project will feature an interface upon which the user is able to adjust the parameters and examine the 2D nature of the triangulation graphically. Due to the two algorithms following different solutions to the triangulation, differences in performance are expected to occur and they are to be tested and reported as part of the project.

Project Environment

We decided to use Python as our programming language and PyCharm for our development environment due to our familiarity with the language and the IDE. We chose PyGame as our graphical user interface framework for the implementation of the Delaunay Triangulation algorithms because of its plotting and similar visualization tools.

Algorithm Descriptions

The brute force algorithm for generating a Delaunay triangulation from a point set is very inefficient and has time complexity of *O(N^3)*. However, there are multiple algorithms that reduce this complexity to *O(NlogN)* time.

One of these is the randomized incremental algorithm which forms a triangulation from the base case and updates the triangulation point by point where the points are randomized meaning no three collinear or co-circular points in the space. Another is the planar divide and conquer algorithm which uses the same principle as mergesort where it recurses to the base case of the triangulation and merges two triangulated graphs.

The algorithms are further inspected and analyzed during the implementation phase. A general overview of the algorithms is that the divide-and-conquer algorithm is faster whereas the incremental algorithm is more flexible because it's incremental, and therefore can be used in an on-line manner by mesh generators and other algorithms that choose vertices based on the state of the triangulation.

1. The Randomized Incremental Algorithm

The randomized incremental algorithm updates the Delaunay Triangulation by incrementally adding a vertex at each step. Addition of a new vertex to the triangulation requires partitioning the Delaunay faces that contain the new vertex to create new faces. The algorithm is expected to have an average runtime of *O(NlogN)* by using the connectivity of the triangulation to efficiently locate affected triangles to remove or add edges. However, the runtime can go up to *O(N^2)* for degenerate point sets. On the other hand, it is still a great improvement to the brute force approach.

The Bowyer-Watson Algorithm is an incremental algorithm where edges are legalized by removing all the edges of triangles whose circumcircle includes the new point. Then, forming new edges from the new point to the vertices of the star-shaped polygon leftover by the edge removal. This way the Delaunay condition is guaranteed to hold with each point addition.

The steps of the Bowyer-Watson Algorithm:

1. Establish a "super"-triangle where all points in the dataset would stay in the interior.
2. Insert a point. Find all the circumcircles and their respective triangles including the new point.
3. Form a polygon from the convex hull of the combined triangles whose interior edges are removed.
4. Insert new edges to every corner of the polygon from the new data point.
5. Repeat steps 2 to 5 for every point in the data set.
6. Remove the points of the super-triangle and any edge using such points.

```
function BowyerWatson (pointList)
   // pointList is a set of coordinates defining the points to be triangulated
   triangulation := empty triangle mesh data structure
   add super-triangle to triangulation // must be large enough to completely contain all the points t
   for each point in pointList do // add all the points one at a time to the triangulation
      badTriangles := empty set
      for each triangle in triangulation do // first find all the triangles that are no longer valid
         if point is inside circumcircle of triangle
            add triangle to badTriangles
      polygon := empty set
      for each triangle in badTriangles do // find the boundary of the polygonal hole
         for each edge in triangle do
            if edge is not shared by any other triangles in badTriangles
               add edge to polygon
      for each triangle in badTriangles do // remove them from the data structure
         remove triangle from triangulation
      for each edge in polygon do // re-triangulate the polygonal hole
         newTri := form a triangle from edge to point
         add newTri to triangulation
   for each triangle in triangulation // done inserting points, now clean up
      if triangle contains a vertex from original super-triangle
         remove triangle from triangulation
   return triangulation
```

Figure 1: Pseudocode for Bowyer-Watson Algorithm

A pseudocode or the Bowyer-Watson algorithm is given in Figure 1 for a more detailed explanation. Images below are provided for visualization of the Bowyer-Watson edge legalization algorithm.

- Red: Polygon, new point
- Green: Edges of new triangles
- Yellow Circle: Illegal circumcircle
- Gray Circle: Legal Circumcircle
- Dotted Blue: Deleted edges of Bad Triangles

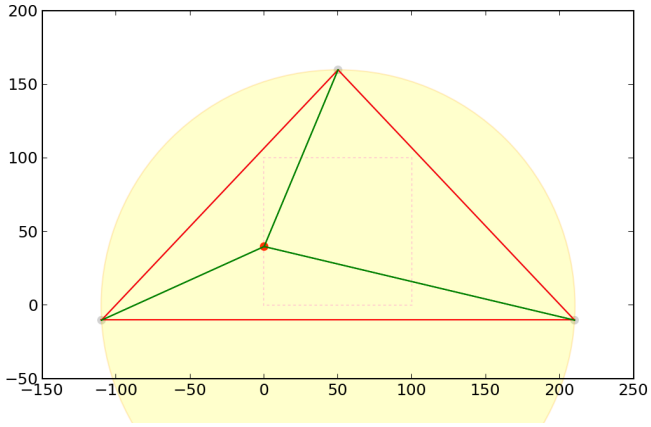


Figure 2: First point addition to the super-triangle.

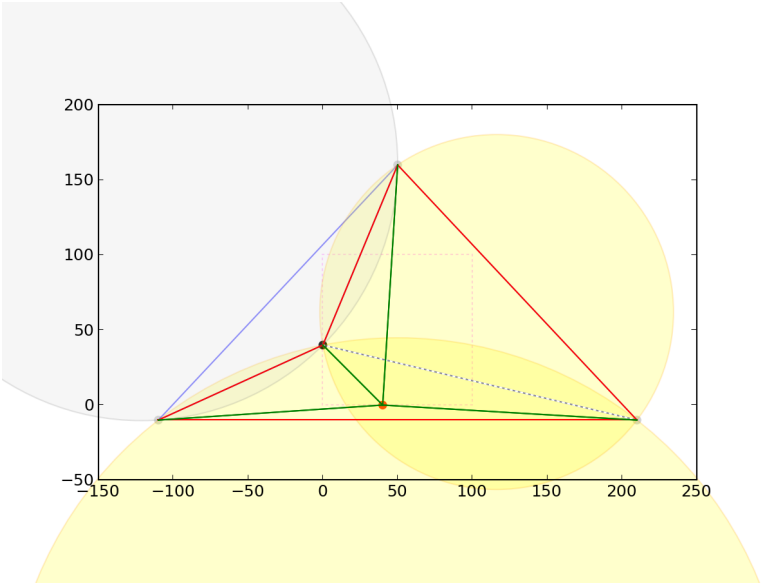


Figure 3: Second point addition into the star-shaped red polygon formed by the triangles of yellow circumcircles.

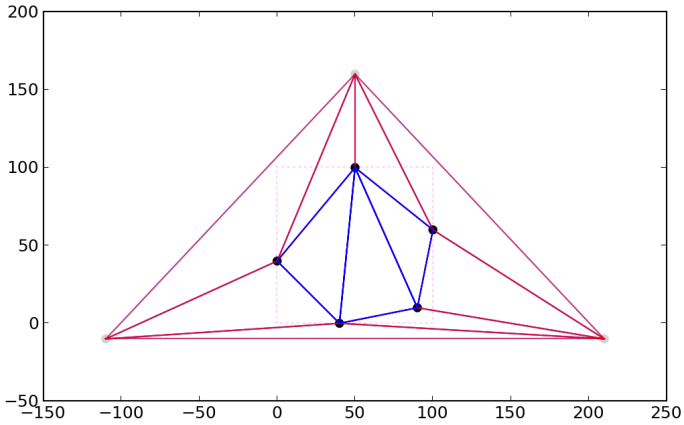


Figure 4: The purple edges forming the Delaunay triangulation and the pink edges discarded due to their connectivity to the super-triangle.

2. The Planar Divide-And-Conquer Algorithm

The algorithm uses splitting lines to recursively divide the set of vertices in half and compute the Delaunay Triangulations separately to then merge them back again. The algorithm is expected to have a worst-case runtime of *O(NlogN),* however for some specific distributions the algorithm may run on expected time of *O(NloglogN)* or even *O(N)*.

Algorithm Steps:

1. Order the points by their x-coordinates. In the case of a tie use y-coordinate as a tie-breaker

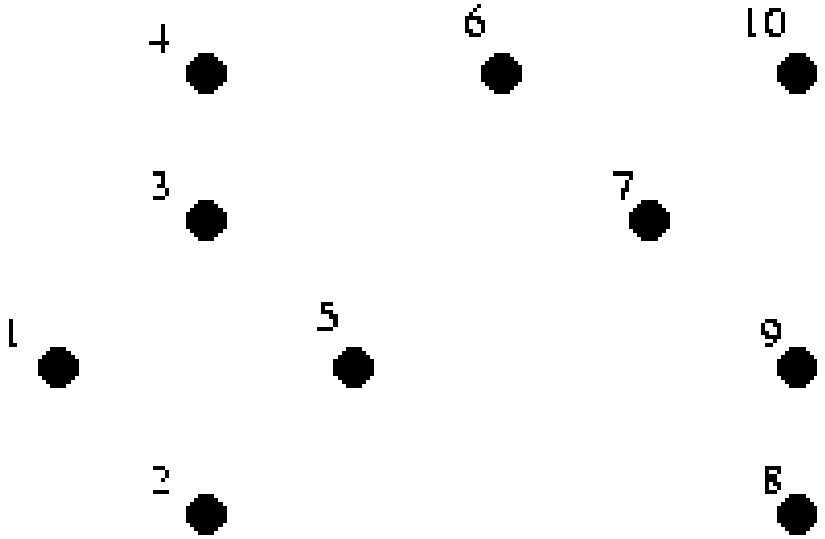


Figure 5: All data points in the plane.

2. Make line segments and triangles of the vertices if the set is of length 2 or 3.
3. Divide the point set in half recursively. The stop condition for each subset is reaching a set of 3 or less points. This results in sets of points that form either a triangle (when the stop condition is reached with 3 points) or a line segment (when the stop condition is reached with 2 points).

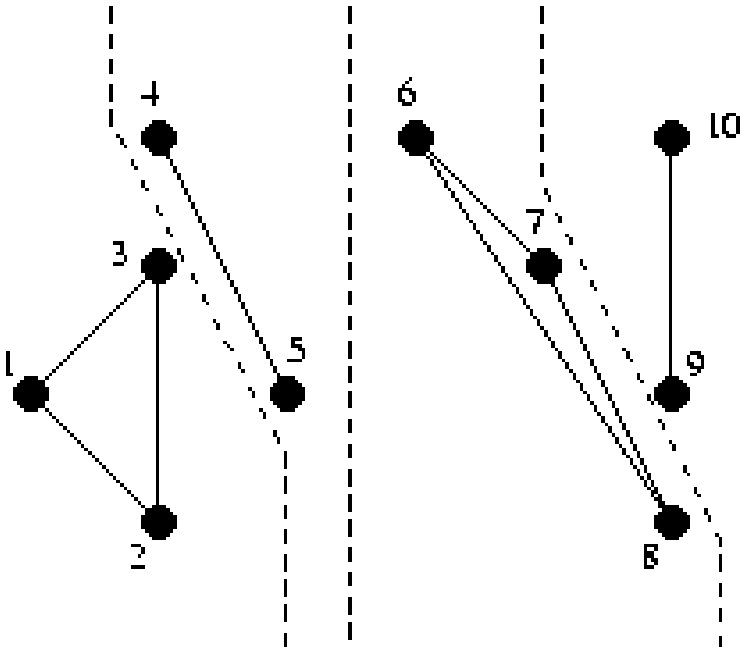


Figure 6: Recursion to base cases.

4. Merge two subdivisions and return their “left-edge” and “right-edge” which are the counterclockwise convex hull edge out of the leftmost vertex and the clockwise convex hull edge out of the rightmost vertex, respectively.

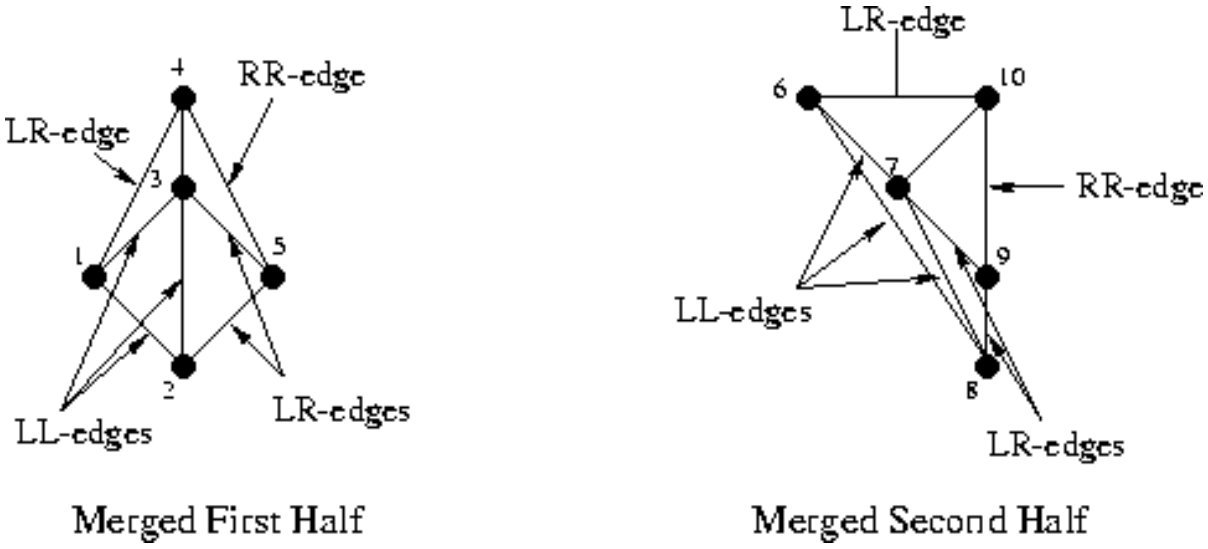


Figure 7: Two merging subdivisions.

The merge operation is intuitive when none of the newly drawn edges (LR-edges) violate the Delaunay property of the triangulation. However the operation is more tricky. The steps of the merge operation are as follows:

4.1. Compute the upper common tangent of the left and right subsets.

4.2. Create a cross edge base using the tangent

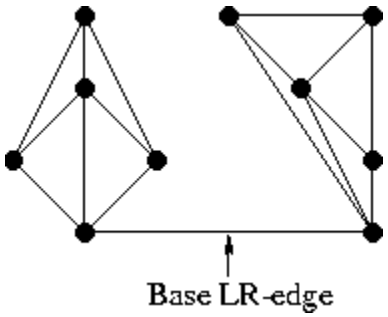


Figure 8: Initial Base LR-edge for merging.

4.3. Until the base is the lower common tangent:

4.3.1 Set left_candidate and right_candidate as the first L and R points to be encountered.

4.3.1. Delete R edges out of right vertex of the base that fail the circle test

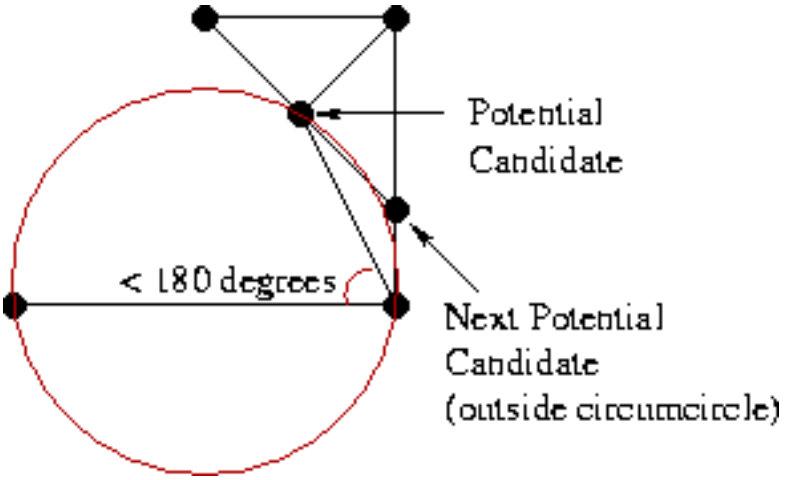


Figure 9: Initial candidate point.

4.3.2. Delete L edges out of left vertex of the base that fail the circle test

4.3.3. If a single candidate satisfies the the following conditions:

4.3.3.1. The clockwise angle from the base LR-edge to the potential candidate must be less than 180 degrees.

4.3.3.2.The circumcircle defined by the two endpoints of the base LR-edge and the potential candidate must not contain the next potential candidate in its interior.

Then it automatically defines the LR-edge to be added

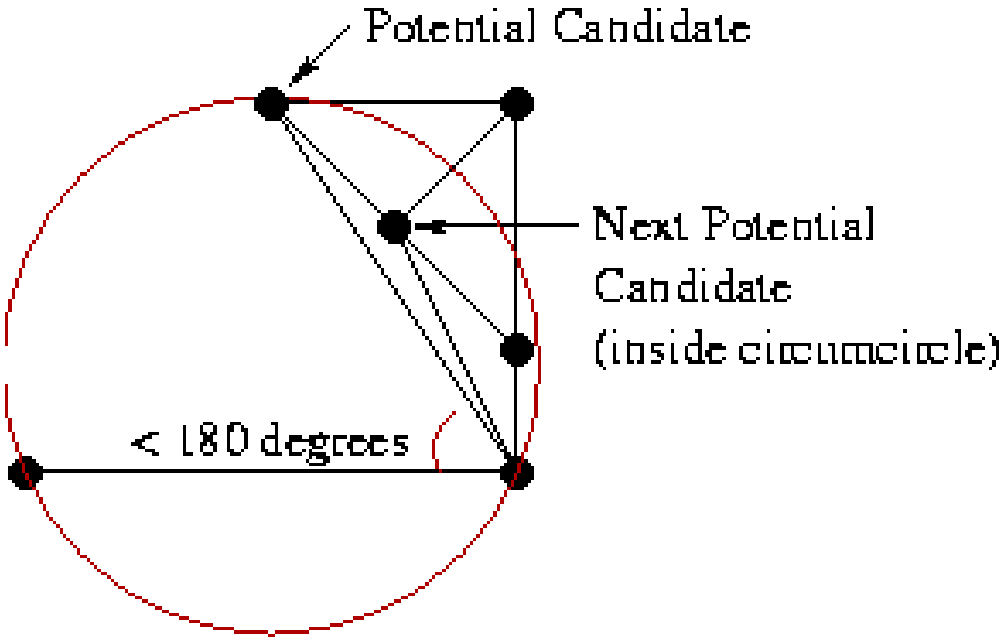


Figure 10: Next and chosen candidate point.

Else if the right candidate is not contained in interior of the circle defined by the two endpoints of the base LR-edge and the left candidate, then the left candidate defines the LR-edge and vice-versa

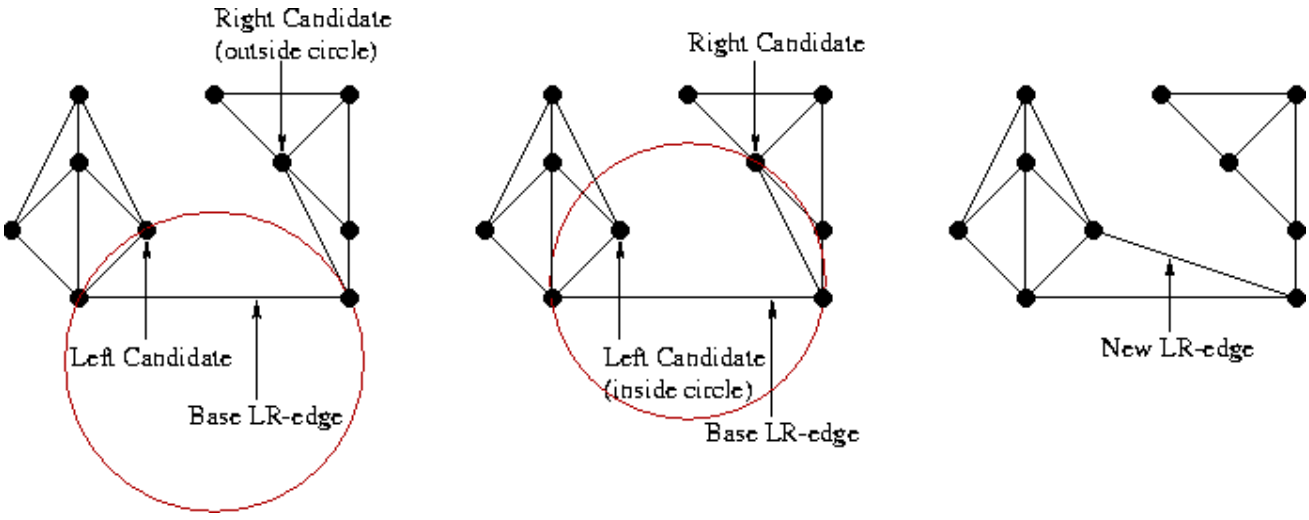


Figure 11: Forming of new LR-edge

4.3.4 Choose the next candidate points with the new LR-edge as the base LR-edge for the next iteration of the loop

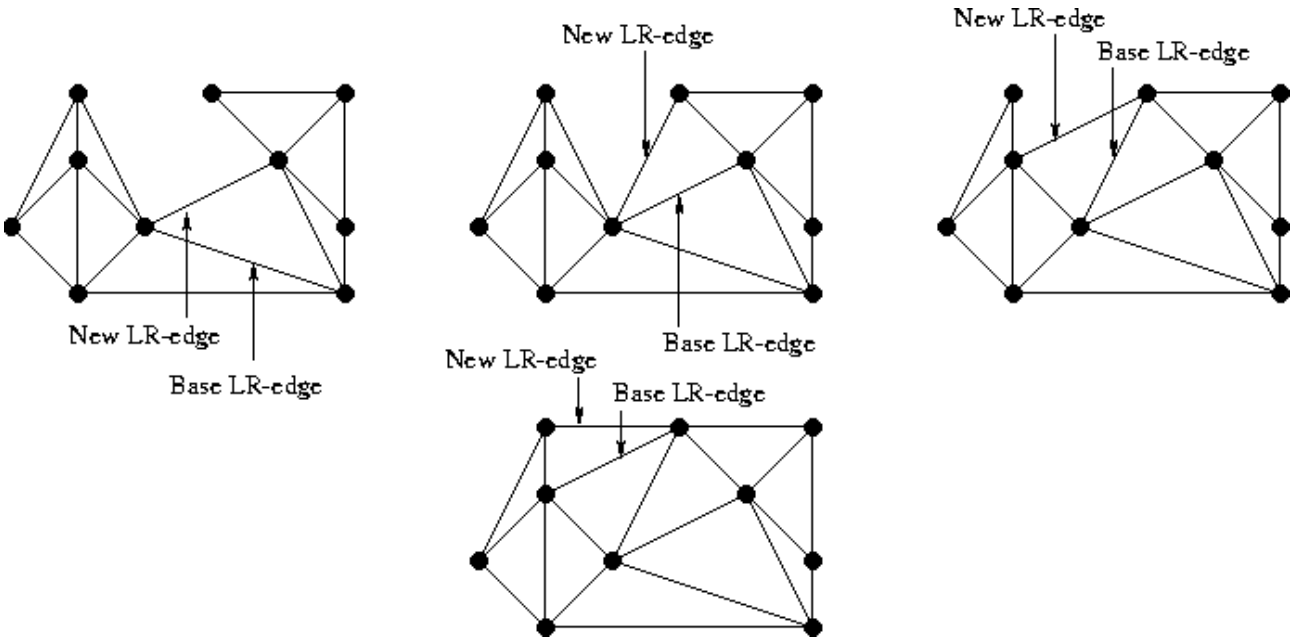


Figure 12: Completion of a single merge.

Thus the end result of the algorithm is:

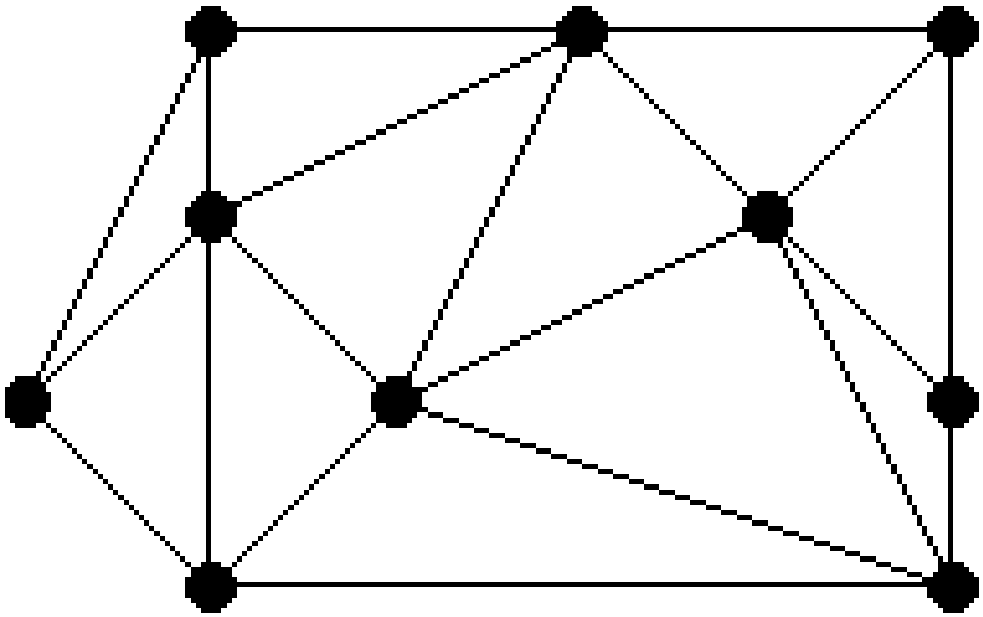


Figure 13: Delaunay Triangulation result of the algorithm.

Used Data Structures

The structures explained in this section are Point, Circle, Triangle and Quad Edge. The Incremental algorithm represents the Delaunay triangulation by holding a list of points, circles, and triangles whereas the divide-and-conquer algorithm keeps a quad edge list.

Bowyer-Watson Algorithm: Circle, Triangle, Point

In order to achieve desired O(NlogN) time complexity from the Bowyer-Watson Algorithm, we would need to drop the search for finding the circles enclosing a point into O(logN) which can be achieved with a range tree which would provide the intersection of different circumcircles that include the new point. However, we implemented a basic Bowyer-Watson algorithm with more fundamental data structures such as a point, circle, and triangle.

The points have the point set of the graph and triangles provide the three edges of all triangles of the graph. The circle array is used to facilitate point inclusion tests for each triangle. The implementation details of the data structures are given below.

Point: (x: float, y: float)
Triangle: (p1: Point, p2: Point, p3: Point)
Circle: (center: Point, radius: float)

Divide and Conquer: Quad Edge

Quad edge is an important data structure as it stores information of neighbors and direction which are very crucial for reducing time complexity in computational geometry. Hence, using a quad-edge structure, iterating through the topology becomes efficient.

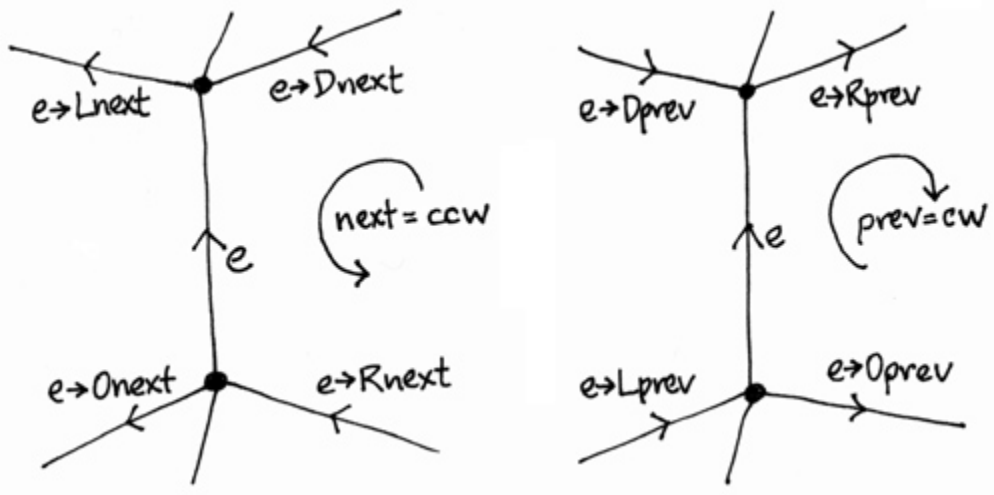


Figure 14: The Quad-Edge Data Structure

Our quad edge implementation holds 6 variables given below. The org and dest variables hold

Points and the edge is directed from org to dest. Since orderings are in counterclockwise directions, “onext” is the next edge right, and “oprev” is the left edge previous to the current edge. The sym variable holds the symmetrical counterpart of the current edge which is very useful for the divide and conquer algorithm because it allows us to apply algorithms that require the edges to be in the correct direction without having to calculate, store and manipulate the directions of edges. For example, in the merging part of the algorithm, candidate vertices are searched to base the merge on. The candidate search operation for the right subdivision requires a clockwise search whereas the same operation for the left subdivision requires a counter-clockwise search. Furthermore, a right subdivision can become the left subdivision of a recursively higher level merge operation. Therefore, using the “sym” variable allows us the flexibility to perform such operations freely. The “data” variable is used to indicate whether an edge has been invalidated or not.

```
self.org   = org
self.dest  = dest
self.onext = None  # right is next (in CCW)
self.oprev = None  # left is previous
self.sym   = None
self.data  = None
```

Implementation Details

The algorithms are already described in detail in the Algorithm Descriptions section. Therefore, this section is reserved only to show the code snippets highlighting the implementation of the algorithm and showing the screenshots of the produced user interface.

1. The Randomized Incremental Algorithm

We required two helper methods for finding the circumcenter of a triangle and checking whether a point was inside a circumcircle. The latter was very simple:

```
def inCircleTest(self, tri, p):
    center, radius = self.circles[tri]
    return np.sum(np.square(center - p)) <= radius
```

The *O(N)* add method is called for every point in the dataset causing a *O(N^2)* time complexity:

```
# Bad Triangles = triangles whose circumcircle contains p
bad_triangles = []
for T in self.triangles:
    if self.inCircleTest(T, p):
        bad_triangles.append(T)
```

Figure 15: Implementation of the linear search for finding bad triangles

```
# Compute the boundary polygon of bad triangles
boundary = []
T = bad_triangles[0]
edge = 0
# get the opposite triangle of this edge
while True:
    # The opposite triangle
    tri_op = self.triangles[T][edge]
    if tri_op not in bad_triangles:
        # Next edge in current triangle
        boundary.append((T[(edge + 1) % 3], T[(edge - 1) % 3], tri_op))
        edge = (edge + 1) % 3
        # Check for loop
        if boundary[0][0] == boundary[-1][1]:
            break
    else:
        # Next edge in the opposite triangle
        edge = (self.triangles[tri_op].index(T) + 1) % 3
        T = tri_op
```

Figure 16: Implementation of finding the boundary polygon of the bad triangles

```
# Delete illegal triangles
for T in bad_triangles:
    del self.circles[T]
    del self.triangles[T]

# Triangulate the polygon
new_triangles = []
for (e0, e1, tri_op) in boundary:
    T = (idx, e0, e1)
    self.circles[T] = self.circumcenter(T)
    self.triangles[T] = [tri_op, None, None]

    # Set T as neighbour of opposite edge if a neighbor is on the same edge
    if tri_op:
        for i, neigh in enumerate(self.triangles[tri_op]):
            if neigh:
                if e1 in neigh and e0 in neigh:
                    self.triangles[tri_op][i] = T

    new_triangles.append(T)
N = len(new_triangles)
for i, T in enumerate(new_triangles):
    self.triangles[T][1] = new_triangles[(i + 1) % N]  # Next neighbour
    self.triangles[T][2] = new_triangles[(i - 1) % N]  # Previous neighbour
```

Figure 17: Implementation of creating the new edges to the polygon

2. The Planar Divide-And-Conquer Algorithm

The algorithm steps in Python: (They follow the steps 1-4 mentioned in the Algorithm description section)

Step 1.

```
# Sort points by x coordinate, y is a tiebreaker.
S.view(dtype=[('f0', S.dtype), ('f1', S.dtype)]).sort(order=['f0', 'f1'], axis=0)

# Remove duplicates.
dupes = [i for i in range(1, len(S)) if S[i-1][0] == S[i][0] and S[i-1][1] == S[i][1]]
if dupes:
    S = np.delete(S, dupes, 0)
```

Figure 18: Implementation of the sorting step

Step 2.

```
if len(S) == 2:
    a = make_edge(S[0], S[1])
    return a, a.sym

elif len(S) == 3:
    # Create edges a connecting p1 to p2 and b connecting p2 to p3.
    p1, p2, p3 = S[0], S[1], S[2]
    a = make_edge(p1, p2)
    b = make_edge(p2, p3)
    splice(a.sym, b)

    # Close the triangle.
    if right_of(p3, a):
        connect(b, a)
        return a, b.sym
    elif left_of(p3, a):
        c = connect(b, a)
        return c.sym, c
    else:  # the three points are collinear
        return a, b.sym
```

Figure 19: Implementation of the base cases for 2 or 3 points.

Step 3.

```
else:
    # Recursively subdivide S.
    m = (len(S) + 1) // 2
    L, R = S[:m], S[m:]
    ldo, ldi = triangulate(L)
    rdi, rdo = triangulate(R)
```

Figure 20: Implementation of the third step of the divide and conquer algorithm.

Step 4. (including all substeps)

```
# Merge.
while True:
    # Locate the first R and L points to be encountered by the diving bubble.
    rcand, lcand = base.sym.onext, base.oprev
    # If both lcand and rcand are invalid, then base is the lower common tangent.
    v_rcand, v_lcand = right_of(rcand.dest, base), right_of(lcand.dest, base)
    if not (v_rcand or v_lcand):
        break
    # Delete R edges out of base.dest that fail the circle test.
    if v_rcand:
        while right_of(rcand.onext.dest, base) and \
              in_circle(base.dest, base.org, rcand.dest, rcand.onext.dest) == 1:
            t = rcand.onext
            delete_edge(rcand)
            rcand = t
    # Symmetrically, delete L edges.
    if v_lcand:
        while right_of(lcand.oprev.dest, base) and \
              in_circle(base.dest, base.org, lcand.dest, lcand.oprev.dest) == 1:
            t = lcand.oprev
            delete_edge(lcand)
            lcand = t
    # The next cross edge is to be connected to either lcand.dest or rcand.dest.
    # If both are valid, then choose the appropriate one using the in_circle test.
    if not v_rcand or \
       (v_lcand and in_circle(rcand.dest, rcand.org, lcand.org, lcand.dest) == 1):
        # Add cross edge base from rcand.dest to base.dest.
        base = connect(lcand, base.sym)
    else:
        # Add cross edge base from base.org to lcand.dest
        base = connect(base.sym, rcand.sym)

return ldo, rdo
```

Figure 21: Implementation of the fourth step of the divide and conquer algorithm.

User Interface

The user interface allows users to view the formed Delaunay triangulation according to their selected algorithm and their given data size. Furthermore, it is possible to show the triangulation work step by step. Furthermore, the user is able to zoom the graph as they wish to see the connections of the graph better by clicking on the graph.

Some screens from the application are shared below for the incremental algorithm. Since the screen for the Divide and Conquer algorithm has the same interface, it is not repeated in this report.

Figure 22: Welcome screen for the application. The user is expected to select one of the algorithms to observe.

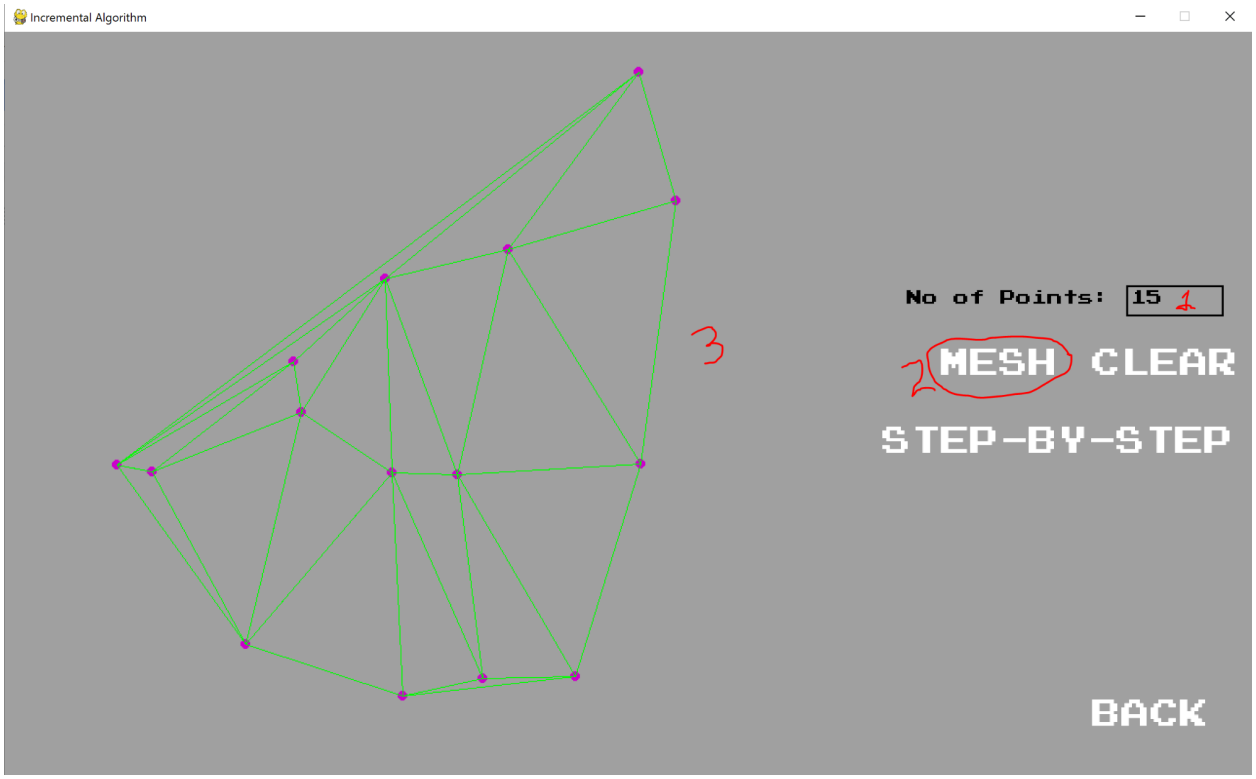


Figure 23: Incremental algorithm screen for the application. The user is expected to select the No of points box to write a number. Then by clicking the MESH button, the output is observed.

The shown graph is cleared by clicking the CLEAR button. The points can also be added one by one by clicking the STEP-BY-STEP button after a clear view.

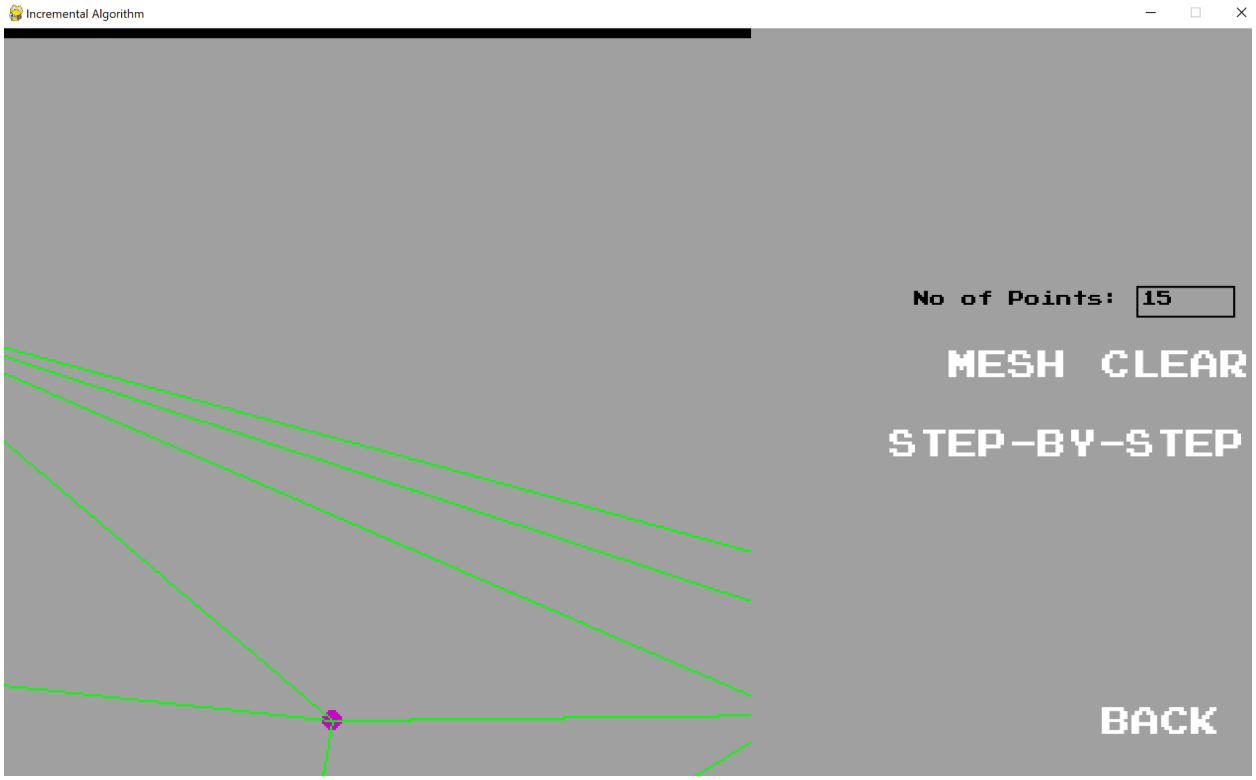


Figure 23: Zoomed in graph view.

By clicking on the graph view, the user will activate and deactivate the zoom feature. The user can freely transform the graph while in zoom with respect to the position of their mouse.

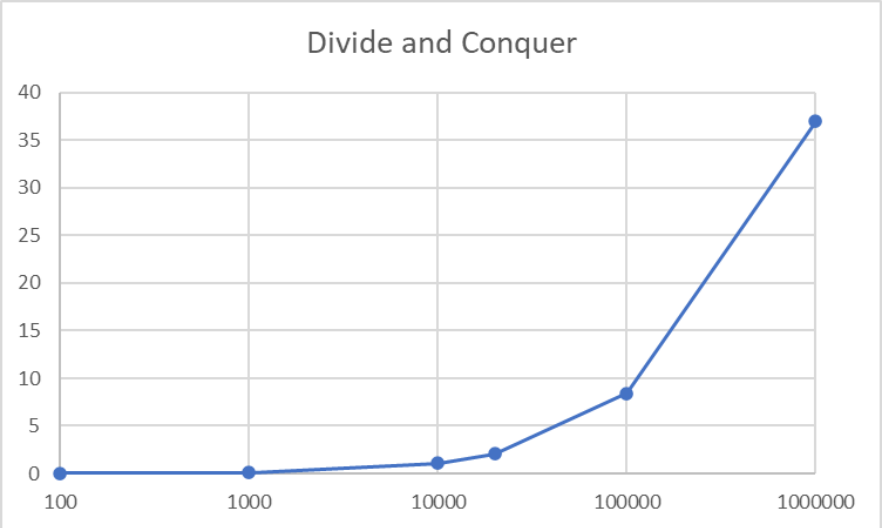
Results

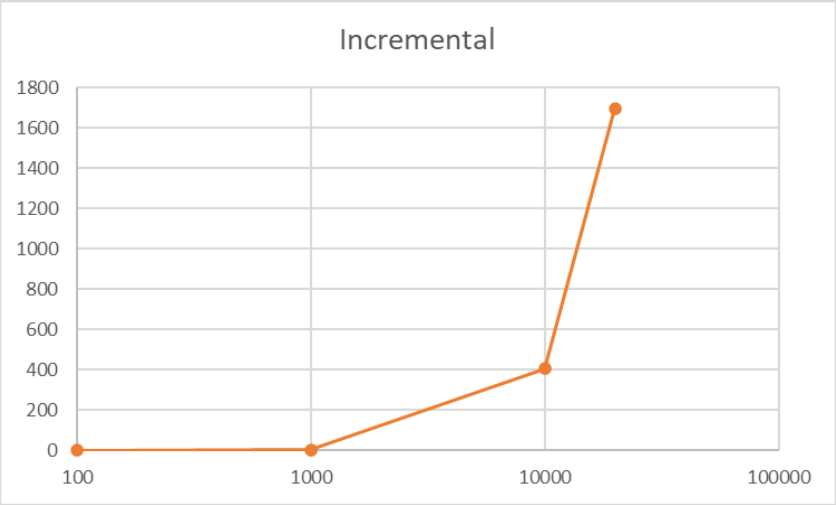
We experimented with the time complexities of our two algorithms. Overall, the divide and conquer algorithm is much faster and more efficient than the incremental one due to the *O(N^2)* implementation of the Bowyer-Watson algorithm.

Set Size	Incremental	Divide And Conquer
100	0.076	0.006
1000	4.214	0.089

10 000	404.3	1.073
20 000	1697	2.080
100 000	Stopped after ~3 hours	8.411
1 000 000	Infeasible	36.95

For the Divide-and-Conquer algorithm, we also observed that in the lower subset sizes the algorithm is so fast that operations that are not exactly part of the triangulation dominate the time. This is seen from the incrementation steps and their outputs. While changing 100 -> 1000 we expect a 10log10 ≈ 23x slow down however the data yielded a 14.8x slow down. Whereas 10000 -> 20000 step is expected to show 2log2 ≈ 1.39 slow down but the results show a 1.94x slow down in our experiment. This indicates that in the lower subset sizes the algorithm is so fast that operations that are not exactly part of the triangulation dominate the time.





References

Abbottjord94. (n.d.). Python-DELAUNAY: A python implementation of Delaunay triangulation. GitHub. Retrieved March 22, 2022, from https://github.com/abbottjord94/python-delaunay

Alexbaryzhikov. (2020, August 11). Triangulation/delaunay.py. GitHub. Retrieved March 22, 2022, from https://github.com/alexbaryzhikov/triangulation/blob/master/delaunay.py

Guibas, Leonidas; Stolfi, Jorge (1985). "Primitives for the manipulation of general subdivisions and the computation of Voronoi". ACM Transactions on Graphics. 4 (2): 74–123. doi:10.1145/282918.282923. S2CID 52852815.

Liu, Yuanxin, and Jack Snoeyink. "A comparison of five implementations of 3D Delaunay tessellation." Combinatorial and Computational Geometry 52 (2005): 439-458

Rebay, S. Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm. Journal of Computational Physics Volume 106 Issue 1, May 1993, p. 127.

Ayron. (n.d.). Ayron/Delaunay: A simple delaunay triangulation library using the Bowyer–Watson algorithm in both python and C++. GitHub. Retrieved May 13, 2022, from https://github.com/ayron/delaunay

Jmespadero. (n.d.). JMESPADERO/pydelaunay2d: A simple Delaunay 2D triangulation in Python (with numpy). GitHub. Retrieved May 13, 2022, from https://github.com/jmespadero/pyDelaunay2D