Algorithmic Analysis and Empirical Evaluation of Delaunay Triangulation Construction

Complexity Crew
Tatva Agarwal
Neharika Rajesh
Shreem Shukla
Kris Jain
Aryan Agrawal

Abstract

This report presents an algorithmic and empirical study of Delaunay Triangulation, focusing on the theoretical foundations, implementation methodologies, and performance evaluation of different construction algorithms. We explore multiple approaches to building the triangulation, analyzing their asymptotic complexities and practical runtime behavior. Experimental results on synthetic and real-world datasets validate theoretical predictions and offer insight into the strengths, limitations, and optimal use cases of each algorithm. Our work provides both a theoretical and empirical understanding of Delaunay Triangulation in computational geometry.

Submission Repository: https://github.com/Neriums05/Complexity-Crew-AAD-Project

Contents

1 Introduction

1.1 Problem Definition

For a given set of points P in a two-dimensional plane.
Given a set of points, there are many ways to connect them to form a mesh of triangles. However, not all triangulations are equal. Some produce long, thin "sliver" triangles that are undesirable for numerical analysis or graphics. The Delaunay Triangulation is the optimal solution that avoids these slivers.
The problem can be defined by the following characteristics:

- **Input:** A set of discrete points $P = \{p_1, p_2, ..., p_n\}$ in $\mathbb{R}^2$.
- **Output:** A triangulation $DT(P)$ such that no point in P is inside the circumcircle of any triangle in $DT(P)$.

Why this is the "Best" Triangulation

Mathematically, Delaunay Triangle satisfies the **Max-Min Angle criterion**: among all possible triangulations of the point set, the Delaunay Triangulation maximizes the minimum angle of all the triangles. This ensures that the triangles are as equilateral ("fat") as possible, rather than thin and stretched (sliver triangles).

1.2 Real-world Relevance

- **Mesh generation:** DT is widely used to create **high-quality meshes** (triangular grids) for surfaces and volumes, as it inherently avoids "skinny" triangles, which is crucial for numerical stability and accuracy in simulations.
- **GIS and terrain modeling:** It forms the backbone of the **Triangulated Irregular Network (TIN)** model, efficiently representing terrain surfaces by connecting scattered elevation points to calculate topographical properties like slopes, aspects, and volumes.
- **Computer graphics:** It is utilized for **surface reconstruction** from scattered point clouds (e.g., from scanners) and for generating efficient triangular meshes for rendering, especially in creating Level of Detail (LOD) representations.
- **Finite element analysis (FEA):** Delaunay meshes provide an ideal **geometric structure** for dividing a complex physical domain into smaller, simpler elements for solving partial differential equations that model physical phenomena like stress, fluid dynamics, or heat transfer.
- **Nearest neighbor search:** The geometric dual of the Delaunay triangulation, the **Voronoi diagram**, inherently partitions space such that finding the closest point to any query location becomes a fast traversal problem through the tessellation structure.

1.3 Objectives

- Implement multiple Delaunay Triangulation algorithms.
- Provide theoretical complexity analysis.
- Experimentally evaluate their performance.
- Compare empirical behavior with theoretical expectations.

2 Algorithm Descriptions

2.1 Flipping Algorithm

2.1.1 Algorithm Introduction: Local Edge Flips

The **edge-flipping algorithm** constructs a Delaunay Triangulation by starting from *any* triangulation of the point set and then locally improving it. The key idea is that the Delaunay property is *local*: a triangulation is Delaunay if and only if every interior edge is *locally Delaunay*. If an interior edge violates this condition, we can *flip* it to increase the minimum angle in the adjacent triangles.

Thus, the algorithm repeatedly finds *illegal* edges and flips them until no further improvement is possible. At that point, all edges are locally Delaunay, and by Lawson's criterion the triangulation is globally Delaunay.

2.1.2 Empty Circumcircle Property and Flipping

(source: link to youtube)
A circle circumscribing any Delaunay triangle does not contain any other input points in its interior. This is the **Empty Circumcircle Property**.

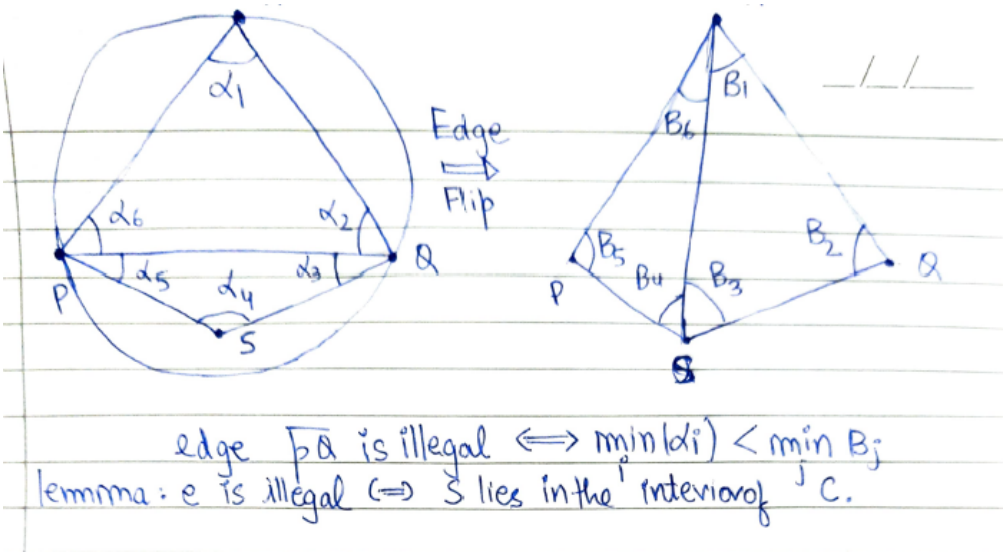


Figure 1

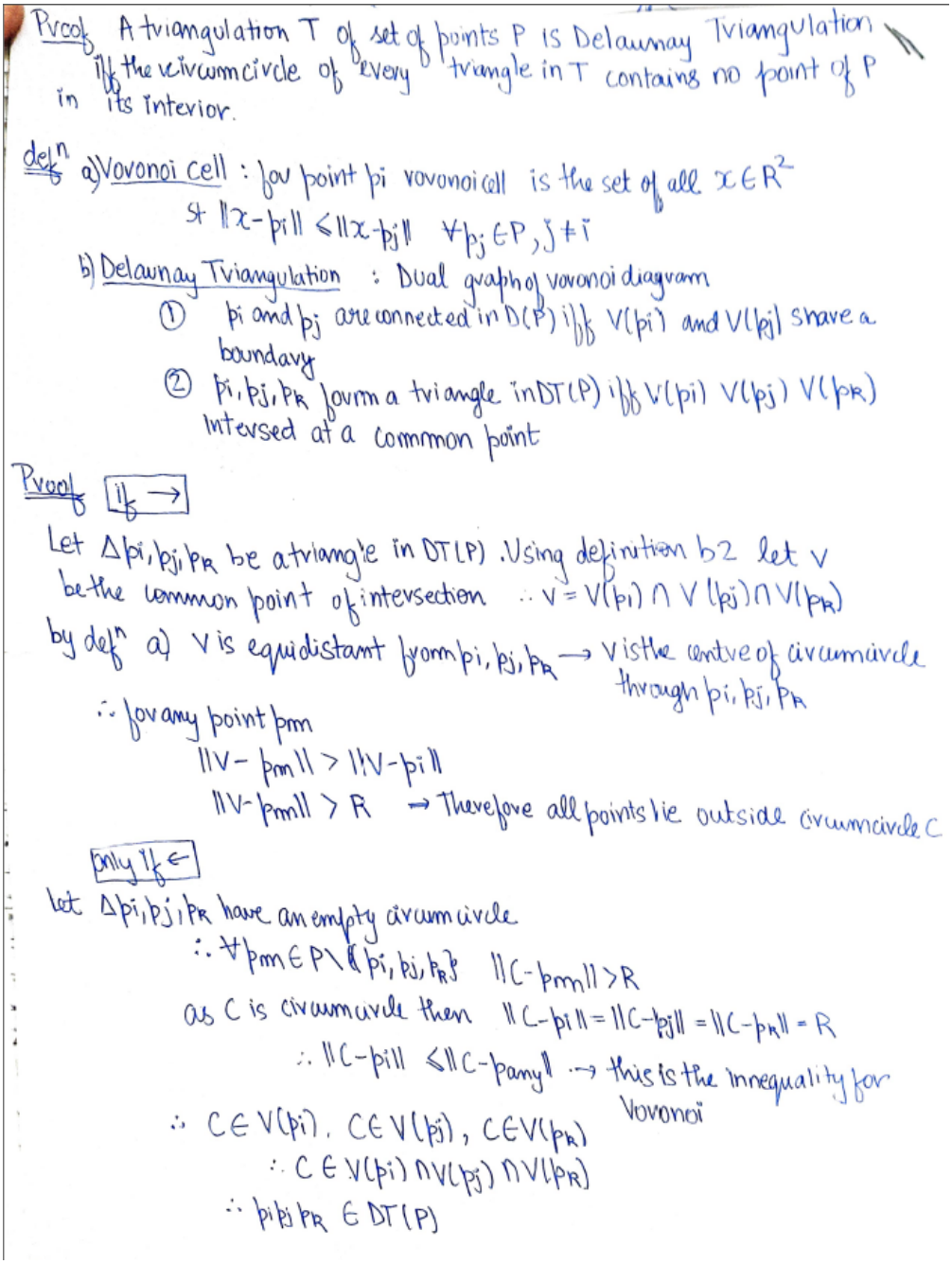


Figure 2

From the above property an important feature arises:

Looking at two triangles ABD, BCD with the common edge BD, if the $\alpha + \gamma \leq 180°$, the triangles meet the Delaunay condition.

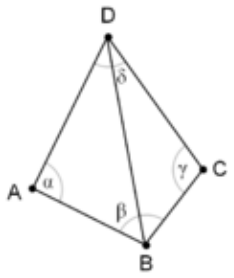


Figure 3

This leads to the important technique called the **flip technique**:
If two triangles do not meet the Delaunay condition, switching the common edge BD for the common diagonal AC produces two triangles that do meet the Delaunay condition.

Formal Algorithm (High-Level)

This insight yields a straightforward algorithm:

1. Construct any initial planar triangulation T of the point set P.
2. For each interior edge e shared by two triangles, test whether e is locally Delaunay (via angle or in-circle test).
3. If e is not locally Delaunay, *flip* e (replace it with the other diagonal of the quadrilateral formed by the two adjacent triangles).
4. Repeat until no edge is illegal.

```
        Global Edge-Flipping Delaunay Triangulation

Input : Point set P in R^2
Output: Delaunay triangulation DT(P)

1. Build any initial triangulation T of P
   (e.g., incremental triangulation without enforcing Delaunay)

2. Initialize a list or queue E of all interior edges of T

3. While E is not empty:
       take an edge e = (b, d) from E
       if e is an interior edge shared by triangles (a, b, d)
          and (b, c, d):
              if e is NOT locally Delaunay:
                  flip e to the other diagonal (a, c)
                  update T
                  add affected neighboring edges back into E

4. Return T
```

2.1.3 Orientation Test

For three points a, b, c, the signed area

$$\text{orient}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

indicates whether c lies to the left or right of the directed segment ab. It is used to maintain consistent triangle orientation and to determine the convexity of the quadrilaterals before flipping.

2.1.4 In-Circle Test

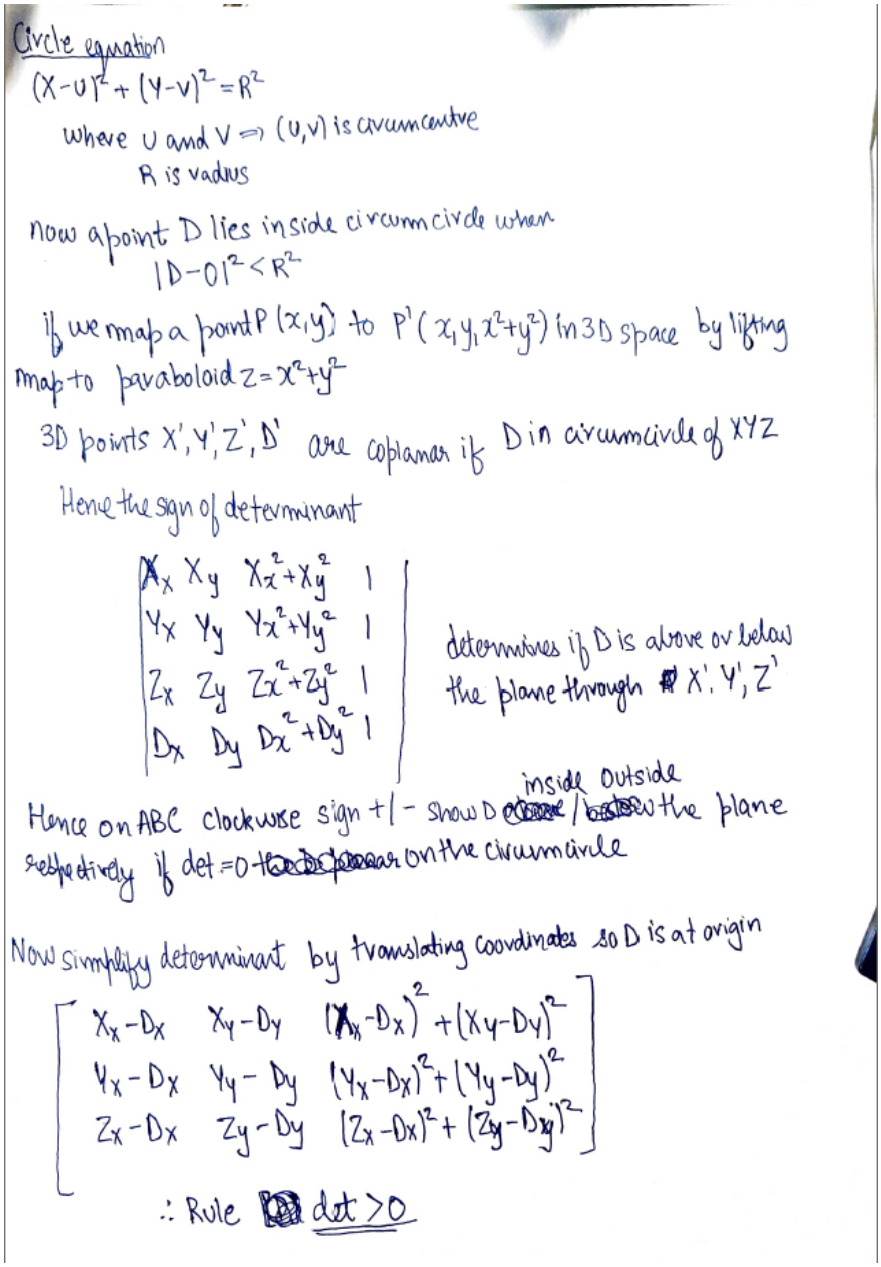


Figure 4

2.1.5 Time Complexity Analysis

Let $n = |P|$ be the number of input points.

Initial Triangulation The running time depends on how we construct the initial triangulation:

- A naive incremental triangulation (searching linearly for a triangle containing each point) can take $O(n^2)$.

In our simple implementation, we use a straightforward incremental scheme, so the initial triangulation is $O(n^2)$ in the worst case.

Edge Flipping Phase Let m be the number of edges (and triangles) in the triangulation; for planar triangulations we have $m = O(n)$.

- Each flip is a constant-time operation (updating a small, fixed number of triangles and adjacency pointers).
- Each flip strictly increases the minimum angle in the local configuration (Lawson's angle maximization argument), and there are only finitely many distinct triangulations of P, so the number of flips is finite.
- Theoretical worst-case bounds give $O(n^2)$ flips in pathological cases, but for random point sets the expected number of flips is $O(n)$.

Putting this together:

$$T(n) = T_{\text{build}}(n) + T_{\text{flips}}(n)$$

- With a naive initial triangulation:

$$T_{\text{build}}(n) = O(n^2), \quad T_{\text{flips}}(n) = O(n^2) \text{ worst case,}$$

so overall $T(n) = O(n^2)$.

- With an $O(n \log n)$ initial triangulation (e.g., S-hull) and typical inputs, the flip phase behaves close to $O(n)$ in practice, so the total time is dominated by $O(n \log n)$.

For the purposes of our implementation and experiments, we report the simple bound:

$$\boxed{\textbf{Total Time Complexity (our implementation):}\ O(n^2)}$$

while noting that better asymptotic behavior is achievable with optimized data structures and initial triangulations.

2.1.6 Space Complexity Analysis

The algorithm maintains:

- The set of input points: $O(n)$.
- A triangulation data structure: $O(n)$ triangles and edges (by Euler's formula for planar graphs).
- A queue or list of candidate edges to test and possibly flip: at most $O(n)$ at any moment.

Therefore, the total memory usage grows linearly with the number of points:

$$\boxed{\textbf{Total Space Complexity:}\ O(n)}$$

2.1.7 Proof Sketch of Correctness

The correctness of the flipping algorithm follows from the *local Delaunay criterion* and Lawson's theorem

Theorem 1 *Starting from any triangulation of P, the edge-flipping algorithm terminates and the final triangulation is the Delaunay Triangulation of P.*

[Proof Sketch]

1. **Local to Global:** Lawson's criterion states that a triangulation is Delaunay if and only if all interior edges are locally Delaunay (no opposite vertex lies inside the neighboring triangle's circumcircle).

2. **Effect of a Flip:** Flipping an illegal edge strictly increases the smallest angle in the two affected triangles (angle maximization lemma). Hence each flip *improves* the triangulation in a well-defined sense.

3. **Termination:** There exist only finitely many distinct triangulations of a fixed point set. Since each flip strictly increases the minimum angle (in a lexicographic ordering of all triangle angles), the process cannot continue indefinitely; thus it terminates after a finite number of flips.

4. **Delaunay at Termination:** When the algorithm stops, no edge is illegal, i.e., all edges satisfy the local Delaunay condition. By Lawson's theorem, this implies that the triangulation is globally Delaunay.

Therefore, the algorithm always terminates and returns the Delaunay Triangulation of P.

2.2 Sweep Hull

2.2.1 Algorithm Introduction: Explanation and Intuition

The **S-hull (Sweep-hull) algorithm** is a deterministic, $\mathbf{O(n \log n)}$ algorithm for constructing the 2D Delaunay Triangulation (DT). Its design separates the triangulation task into two intuitive phases:

- A fast **radial sweep** that constructs an initial planar triangulation.
- A **local edge-flipping refinement** step enforcing the Empty Circumcircle Property, resulting in the Delaunay Triangulation.

The central insight is that maintaining global Delaunay constraints during point insertion is computationally expensive. Instead, S-hull imposes structure by radially sorting points around a seed triangle, dramatically simplifying incremental updates. After this initial triangulation is built, the Delaunay property is restored via classical Lawson edge flips.

Stage I: Sweep-Hull Construction

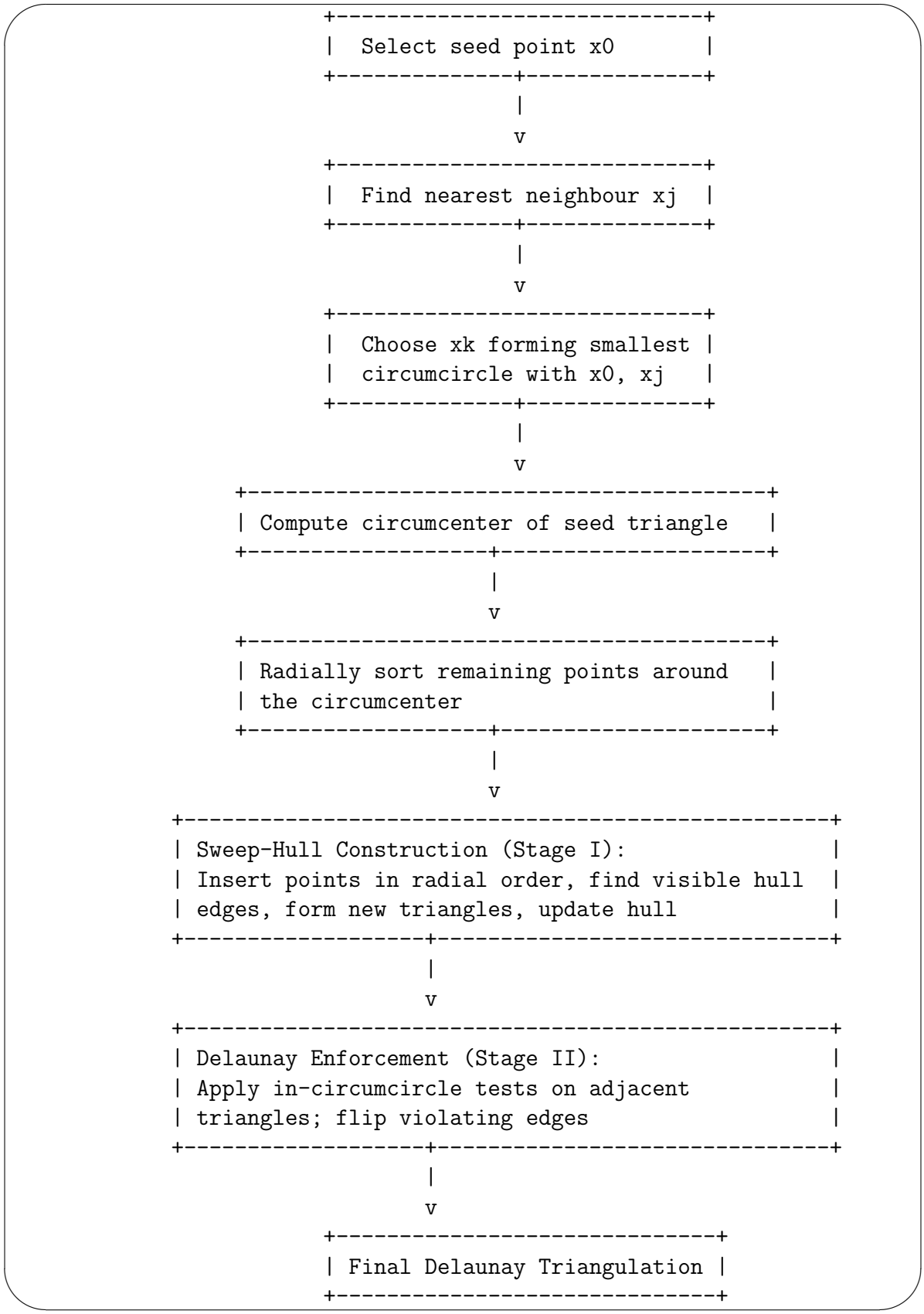
The algorithm begins with selecting a seed triangle, typically chosen to minimize the circumcircle size. The remaining points are radially sorted around the seed's circumcenter. Points are inserted in this order, updating the current convex hull: visible hull edges from the new point are identified, and triangles are formed with these edges.

This results in a consistent planar triangulation but not yet the Delaunay triangulation.

Stage II: Delaunay Enforcement via Edge Flipping

After the sweep phase, each pair of adjacent triangles is tested with the in-circumcircle predicate. If the shared edge fails to satisfy the local Delaunay condition, it is flipped.

Because each flip strictly increases the minimum angle among the affected triangles (Lawson 1977), infinite flips are impossible. When no violating edges remain, the triangulation is globally Delaunay.


Select seed point x0
Find nearest neighbour xj
Choose xk forming smallest circumcircle with x0, xj
Compute circumcenter of seed triangle
Radially sort remaining points around the circumcenter
Sweep-Hull Construction (Stage I):
Insert points in radial order, find visible hull edges, form new triangles, update hull
Delaunay Enforcement (Stage II):
Apply in-circumcircle tests on adjacent triangles; flip violating edges
Final Delaunay Triangulation

2.2.2 Time Complexity Analysis

The runtime is the sum of the three main processes:

$$T(n) = O(T_{\text{sort}}) + O(T_{\text{sweep}}) + O(T_{\text{flips}})$$

- **Radial Sorting:** $O(n \log n)$ — dominant term.
- **Sweep-Hull Construction:** $O(n)$ amortized.
- **Delaunay Enforcement:** $O(n)$ expected; $O(n^2)$ worst-case.

$$\boxed{\textbf{Total Time Complexity: } O(n \log n)}$$

The radial sort dominates the runtime; edge flips grow only linearly in expectation.

2.2.3 Space Complexity Analysis

Using Euler's formula for planar triangulations:

$$V - E + F = 1$$

A Delaunay triangulation has:

$$E = O(n), \quad F = O(n)$$

Thus the full algorithm requires:

$$\boxed{\textbf{Total Space Complexity: } O(n)}$$

Memory usage is linear in the number of points and depends mostly on storage of triangles, edges, and adjacency lists.

2.2.4 Formal Proof of Correctness

Correctness follows from classical geometric results on Delaunay triangulations and the convergence of edge-flip operations, especially Lawson's foundational results [1].

Definition 1 (Delaunay Triangulation (DT)) *A triangulation T of a point set P is a Delaunay Triangulation if the circumcircle of every triangle in T contains no point of P in its interior.*

Theorem 2 (Lawson Criterion (1977) [1]) *A triangulation T is Delaunay if and only if all interior edges satisfy the local Delaunay condition: the opposite vertex of each adjacent triangle pair does not lie inside the circumcircle of the other triangle.*

Proof Sketch

Lemma 1 (Angle Maximization) *Flipping a non-Delaunay edge increases the minimum angle among the six angles in the two adjacent triangles.*

Let $\alpha_{\min}(T)$ be the minimum angle in triangulation T.

1. **Uniqueness:** Under general position (no four cocircular points), the Delaunay triangulation is unique.
2. **Flip Test:** Edge BC is non-Delaunay iff D lies inside the circumcircle of $\triangle ABC$.
3. **Termination:** Each flip increases $\alpha_{\min}(T)$ strictly; there are finitely many triangulations $\rightarrow$ must terminate.
4. **Correctness After Termination:** By Lawson's result, a triangulation with no illegal edges is globally Delaunay.

Thus the algorithm always terminates and returns the correct DT.

2.3 Bowyer–Watson Algorithm

2.3.1 Algorithm Introduction: Incremental Delaunay via Cavity Retriangulation

The **Bowyer–Watson algorithm** is a classic incremental algorithm for constructing the 2D Delaunay Triangulation. Like the flip-based approaches, it explicitly enforces the *Empty Circumcircle Property*, but instead of flipping edges in an existing triangulation, it incrementally rebuilds local regions affected by each new point.

The high-level idea is:

- Start with a large **supertriangle** that contains all input points.
- Insert points one by one.
- For each new point, remove all triangles whose circumcircles contain that point (the *conflict region* or *cavity*).
- Retriangulate the resulting polygonal hole by connecting its boundary edges to the new point.

This local retriangulation ensures that after each insertion, the triangulation remains Delaunay.

Stage I: Supertriangle Initialization

- Compute the bounding box of the input point set P.
- Construct a supertriangle whose vertices lie far outside this bounding box (for instance, by scaling the box by a factor of 20).
- Initialize the triangulation $\mathcal{T}$ with this single supertriangle.

All input points are guaranteed to lie strictly inside this supertriangle.

Stage II: Incremental Point Insertion

For each point $p_i \in P$ (in any chosen order):

1. **Identify bad triangles:** Scan all current triangles $T \in \mathcal{T}$ and collect those whose circumcircle contains p_i in its interior. These are called *bad triangles*, as they would violate the Delaunay condition after inserting p_i.
2. **Form the polygonal cavity boundary:** Consider all edges of the bad triangles. Edges that belong to exactly one bad triangle form the boundary of the cavity. Edges that are shared by two bad triangles lie strictly inside the cavity and are discarded.
3. **Remove bad triangles:** Remove all bad triangles from $\mathcal{T}$, leaving a polygonal hole.
4. **Retriangulate the cavity:** For each boundary edge e of this polygon, create a new triangle by connecting e to p_i, and add these triangles to $\mathcal{T}$.

Stage III: Supertriangle Cleanup

After all points have been inserted, remove any triangle in $\mathcal{T}$ that uses a vertex of the supertriangle. The remaining triangles form the Delaunay Triangulation of the original point set.

```
            Bowyer--Watson Delaunay Triangulation (2D)

Input : Point set P in R^2
Output: Delaunay triangulation DT(P)

1. Construct a supertriangle T0 that contains all points in P
2. T := { T0 }

3. For each point p in P:
       Bad := empty set
       For each triangle t in T:
           if p lies inside circumcircle(t):
               add t to Bad

       Poly := boundary edges of union of Bad
               (edges that appear in exactly one triangle in Bad)

       Remove all triangles in Bad from T

       For each edge e in Poly:
           Create triangle (e, p)
           Add this triangle to T

4. Remove any triangle in T that has a vertex from the supertriangle

5. Return T
```

2.3.2 Key Geometric Predicates

The algorithm relies on two geometric building blocks:

Orientation Test Given three points a, b, c, the signed area

$$\text{orient}(a, b, c) = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

determines whether c lies to the left or right of the directed line ab. This is used for robust edge and triangle handling.

In-Circle (Circumcircle) Test Given triangle vertices (x_1, y_1), (x_2, y_2), (x_3, y_3) and a query point (x, y), the in-circle predicate answers whether (x, y) lies inside the circumcircle of the triangle. Algebraically, this can be expressed as the sign of a 4×4 determinant; in implementation, one typically uses a stabilized version of this test (with a small ε tolerance) to combat floating-point error.

These predicates are the same building blocks used in flip-based Delaunay algorithms, so many implementation details can be shared between the two methods.

2.3.3 Time Complexity

Let $n = |P|$ be the number of input points.

- **Worst Case:** In the simplest implementation, each point insertion may require checking all existing triangles for the in-circle test. Since a planar triangulation has $O(n)$ triangles, this yields
$$T_{\text{worst}}(n) = O(n^2).$$
Adversarial point orders or highly structured inputs (e.g., grid or circular arrangements) can trigger this behavior.

- **Average Case:** For random point distributions in the plane, the expected size of the conflict region (the set of bad triangles whose circumcircles contain the new point) is small—typically $O(1)$ or $O(\log n)$ in practice. Hence, the expected runtime is
$$T_{\text{avg}}(n) = O(n \log n),$$
comparable to other optimal Delaunay constructions.

- **Best Case:** With a particularly favorable insertion order (e.g., from interior to exterior), each point may only affect a constant number of triangles, leading to
$$T_{\text{best}}(n) = O(n).$$

In our implementation (and most straightforward ones), we use the simple $O(n)$-per-point triangle scan, so we theoretically obtain $O(n^2)$ worst-case complexity, but empirically much closer to $O(n \log n)$ on typical datasets.

2.3.4 Space Complexity

As with any planar triangulation, the number of triangles and edges is linear in n by Euler's formula. The algorithm stores:

- The input point set: $O(n)$.
- The current triangle list: $O(n)$ triangles.
- Temporary containers for bad triangles and cavity boundary edges: $O(n)$ in the worst case but typically much smaller.

Thus the overall space complexity is:

$$\boxed{\textbf{Space Complexity: } O(n)}.$$

2.3.5 Proof Sketch of Correctness

Theorem 3 *At every step of the Bowyer–Watson algorithm, the triangulation $\mathcal{T}$ is Delaunay for the points inserted so far. After all points are inserted and supertriangle vertices are removed, $\mathcal{T}$ is the Delaunay triangulation of P.*

[Proof Sketch] We argue by induction on the number of inserted points.

Base Case: Initially, $\mathcal{T}$ consists only of the supertriangle, which trivially satisfies the Empty Circumcircle Property for its three vertices.

Induction Step: Assume that before inserting point p_i, the triangulation $\mathcal{T}$ is Delaunay for the current point set P_{i-1}. When we insert p_i:

1. *Identification of bad triangles:* Any triangle whose circumcircle contains p_i would violate the Delaunay condition if left unchanged, so all such triangles are collected into the conflict region.

2. *Cavity boundary correctness:* Edges that belong to exactly one bad triangle form the boundary of the cavity. These boundary edges are guaranteed to be visible from p_i and form a simple polygon around p_i.

3. *Retriangulation:* By connecting p_i to each boundary edge, we obtain new triangles whose circumcircles do not contain any previously inserted point, otherwise those points would have already invalidated the Delaunay property earlier (contradiction with the induction hypothesis).

4. *Locality of changes:* Triangles outside the conflict region are unchanged and remain Delaunay, since their circumcircles do not contain p_i by definition.

Thus, after retriangulating the cavity, the entire triangulation is Delaunay for P_i. When all points have been inserted, removing triangles incident to the artificial supertriangle vertices leaves exactly the Delaunay triangulation of the original set P.

2.4 Randomized Incremental Construction (RIC)

The **Randomized Incremental Construction (RIC)** algorithm is an advanced method for computing the Delaunay Triangulation. Unlike naive incremental approaches that can degrade to $O(n^2)$, RIC achieves an expected time complexity of $O(n \log n)$ by inserting points in a random order and maintaining a specialized search structure called a **History DAG**.

2.4.1 Algorithm Phases

The algorithm consists of three main components: Initialization, Point Location, and Structural Updates.

1. Initialization We begin by creating a "Supertriangle" that is large enough to contain all points in the input set P. This ensures that every inserted point falls within an existing triangle, simplifying boundary conditions.

- The Supertriangle is the root of our History DAG.
- The input points P are shuffled randomly. This randomization is crucial for the asymptotic complexity guarantees.

2. Point Location (History DAG) To insert a new point p_r, we must first find the triangle Δ in the current triangulation that contains p_r. A linear search would result in $O(n^2)$ complexity. Instead, we use a History DAG.

Structure of the DAG:

- **Nodes:** Every triangle ever created during the algorithm's execution is a node.
- **Internal Nodes:** "Dead" triangles that have been split or destroyed by an edge flip. They point to the new triangles that replaced them.
- **Leaf Nodes:** "Alive" triangles currently part of the triangulation.

Traversal: To locate point p_r, we start at the root (Supertriangle). If the current node is not a leaf, we check its children. Since the children of a dead triangle form a partition of its area, p_r must lie in exactly one child. We descend the graph until we reach a leaf node.

3. Insertion and Split Once the containing triangle $\Delta(a, b, c)$ is found: 1. We delete $\Delta(a, b, c)$ from the active set. 2. We connect p_r to vertices a, b, and c, creating three new triangles: $T_1(a, b, p_r)$, $T_2(b, c, p_r)$, and $T_3(c, a, p_r)$. 3. In the DAG, the node for Δ is marked as a parent to T_1, T_2, T_3.

4. Legalization (Edge Flipping) The insertion of p_r may violate the Delaunay property locally. We check the edges of the original containing triangle (now the outer edges of the "star" formed by p_r).

For an edge $e = (u, v)$ shared by new triangle $T(u, v, p_r)$ and an adjacent old triangle $T_{opp}(v, u, q)$: 1. We check if q lies inside the circumcircle of T. 2. If it does, the edge e is illegal. We perform an **Edge Flip**:

- The diagonal uv is replaced by $p_r q$.
- Two new triangles are created.
- The DAG is updated: the two old triangles become parents to the two new triangles.

3. This process is recursive. Any new edges created by a flip are also checked until all edges are locally Delaunay.

2.4.2 Pseudocode

Algorithm 1 Randomized Incremental Construction

Input: Set of points P
Output: Delaunay Triangulation $\mathcal{T}$
Initialize DAG with Supertriangle containing P
Randomly shuffle P
for each point $p_r \in P$ **do**
    $\Delta \leftarrow \text{Locate}(DAG, p_r)$ ▷ Descend DAG to find leaf
    Split Δ into T_1, T_2, T_3 connecting vertices of Δ to p_r
    Update DAG: $\Delta \rightarrow \{T_1, T_2, T_3\}$
    for each new external edge e of T_1, T_2, T_3 **do**
        LegalizeEdge(p_r, e, DAG)
    end for
end for
Remove Supertriangle vertices and incident edges
return Leaf nodes of DAG

Algorithm 2 LegalizeEdge

function LegalizeEdge$(p, (u, v), DAG)$
    Let q be the vertex opposite to p across edge (u, v)
    if q exists **and** q is inside circumcircle of $\Delta(u, v, p)$ **then**
        Flip edge (u, v) to (p, q)
        Create new triangles T_a, T_b
        Update DAG: Old pair $\rightarrow \{T_a, T_b\}$
        LegalizeEdge$(p, (u, q), DAG)$
        LegalizeEdge$(p, (q, v), DAG)$
    end if
end function

2.4.3 Asymptotic Analysis

Time Complexity **1. Point Location:** The cost of inserting point p_r is dominated by the time taken to traverse the DAG.

- In the **Worst Case** (e.g., points inserted in sorted order along a line), the DAG can degenerate into a list, leading to $O(n)$ per insertion and $O(n^2)$ total time.
- However, because we **randomly shuffle** P, the structure of the triangulation changes randomly. Guibas, Knuth, and Sharir proved that the expected depth of the DAG is logarithmic.
- Expected time per location: $O(\log n)$.
- Total expected time for location: $\mathbf{O(n \log n)}$.

2. Structural Updates (Flips): Once the point is located, we modify the mesh.

- Splitting a triangle takes $O(1)$.
- The number of edge flips required to restore the Delaunay property corresponds to the number of structural changes in the underlying graph.
- In a planar graph, the average degree of a vertex is bounded (approx. 6).
- Consequently, the **expected total number of edges created and destroyed** during the entire algorithm is $O(n)$.
- Average cost of flipping per point: $O(1)$.

Total Expected Time Complexity:

$$O(n \log n) \text{ (Location)} + O(n) \text{ (Updates)} = \mathbf{O(n \log n)}$$

Space Complexity The space complexity is determined by the size of the History DAG.

- Every triangle created during the algorithm (whether currently alive or dead) is a node in the DAG.
- Since the expected total number of structural changes (splits and flips) is $O(n)$, the expected number of nodes in the DAG is $O(n)$.

Total Expected Space Complexity: O(n).

2.4.4 Correctness and the Role of Randomization

Termination and Delaunay Property It has been established in earlier algorithms the correctness of termination and convergence properties.

1. **Termination:** The flipping process corresponds to optimizing a global functional (lifting the triangulation to a paraboloid). Every flip lowers the surface area of the lifted triangulation. Since the number of possible triangulations is finite, the algorithm must terminate.
2. **Convergence:** A triangulation where every edge is locally Delaunay is equivalent to the global Delaunay Triangulation. Upon termination, no illegal edges remain, guaranteeing the correct output.

Why Randomization is Necessary While the standard flipping algorithm is correct regardless of insertion order, its efficiency is highly sensitive to that order. Without randomization, the algorithm can degrade to $O(n^2)$. Randomization ensures the $O(n \log n)$ bound through two mechanisms:

1. Bounding the DAG Depth: If points are inserted in a sorted order (e.g., along a line), the History DAG becomes unbalanced, degenerating into a linked list with depth $O(n)$.

- By shuffling the points, we ensure that the triangulation evolves "evenly" across the domain.
- Using **Backwards Analysis**, the probability that the i-th inserted point significantly alters the search structure for previous points is small.
- This guarantees that the expected depth of the DAG remains $O(\log n)$.

2. Bounding Structural Changes: The cost of inserting the r-th point includes the number of edges flipped.

- In the worst case, inserting a single point can trigger $O(r)$ flips (e.g., creating a "wagon wheel" that reconnects to all boundary points).
- However, in a random order, the expected degree of a new vertex in the triangulation of r random points is constant (specifically, 6 by Euler's formula for planar graphs).
- Therefore, the expected number of pointers created or destroyed in step r is $O(1)$. Summing over n insertions gives $O(n)$ total structural work.

2.5 Divide and Conquer

2.5.1 Algorithm Introduction

The **Divide and Conquer** algorithm for Delaunay Triangulation was developed by *Leonidas Guibas* and *George Stolfi* in 1985 [file:1]. The algorithm achieves $O(n \log n)$ time complexity by recursively decomposing the problem into smaller subproblems, computing the Delaunay Triangulation on subsets of points, and then merging them using cross edges while maintaining the Delaunay property [file:1].

The key insight is that we can split the point set, recursively triangulate each half, and merge the results efficiently while preserving the empty circumcircle property [file:1].

2.5.2 Algorithm Overview

The algorithm proceeds in the following phases:

1. Sorting The input points are sorted by x-coordinate (with y-coordinate as tiebreaker) [file:1]. This establishes the x-axis as the basis for splitting the data into halves in subsequent steps [file:1].

2. Base Case If there are 3 or fewer points, the triangulation is trivially constructed [file:1]. For 2 points, we create a single edge; for 3 points, we create a triangle [file:1].

3. Divide The sorted points are split into two sets L and R based on their x-coordinate and a threshold (the median point) [file:1].

4. Conquer Recursively compute Delaunay triangulations $D(L)$ and $D(R)$ for the left and right subsets [file:1].

5. Merge This is the most critical phase that enables the divide-and-conquer approach [file:1]. Given $D(L)$ and $D(R)$ as disjoint triangulations without edges crossing between them, we must merge them efficiently [file:1].

2.5.3 Merge Algorithm (Detailed)

The merge phase consists of two main steps:

Step 1: Finding and Adding the Base Edge (Lower Common Tangent) The base edge (L_0, R_0) connects a point L_0 from the left convex hull to a point R_0 from the right convex hull such that all other points lie above or on the line through L_0 and R_0, and the edge doesn't intersect any existing edges from $D(L)$ or $D(R)$ [file:1].

Algorithm to find the base edge:

1. **Start with candidates:** Let L_{curr} be the rightmost point on the left convex hull (maximum x-coordinate in left set) [file:1]. Let R_{curr} be the leftmost point on the right convex hull (minimum x-coordinate in right set) [file:1].

2. **Move left pointer down:** While there exists a point L_{next} that is the clockwise neighbor of L_{curr} on the left convex hull such that the triple $(L_{\text{next}}, L_{\text{curr}}, R_{\text{curr}})$ makes a **right turn** (negative cross product), update $L_{\text{curr}} = L_{\text{next}}$ [file:1].

   Why this works: A right turn means L_{next} is below the line $(L_{\text{curr}}, R_{\text{curr}})$. Moving to L_{next} lowers the potential base edge [file:1].

3. **Move right pointer down:** While there exists a point R_{next} that is the counter-clockwise neighbor of R_{curr} on the right convex hull such that the triple $(L_{\text{curr}}, R_{\text{curr}}, R_{\text{next}})$ makes a **right turn**, update $R_{\text{curr}} = R_{\text{next}}$ [file:1].

4. **Iterate:** Repeat steps 2-3 until no further movement is possible [file:1]. At this point, $(L_{\text{curr}}, R_{\text{curr}})$ is the base edge (L_0, R_0) [file:1].

Mathematical correctness: The algorithm terminates because each iteration strictly lowers at least one endpoint (in terms of y-coordinate, or if y is the same, by geometric position) [file:1]. The final edge is the unique lower tangent because no points lie below it [file:1].

Time complexity: $O(n)$ because each point on the convex hull is examined at most twice (once from each direction) [file:1].

Step 2: Adding Cross Edges Iteratively Once we have the base edge, we build the remaining cross edges by moving upward from bottom to top [file:1]. This is the most intricate part [file:1].

Invariant: At each step, we have a current base edge (L_i, R_i) that is known to be in the final Delaunay triangulation [file:1]. We seek the next cross edge (L_{i+1}, R_{i+1}) above it [file:1].

Key insight: The next cross edge must have either $L_{i+1} = L_i$ (same left endpoint) or $R_{i+1} = R_i$ (same right endpoint), but not both [file:1]. This is because cross edges form a monotone chain from bottom to top [file:1].

Finding Candidate Points For the current base edge (L_i, R_i), we identify two types of candidates:

- **Left candidate** L_{cand}:
  - *Definition:* The first point encountered when moving **counter-clockwise** from L_i around the left convex hull that forms a valid potential cross edge with R_i [file:1].
  - *Geometric meaning:* Among all edges in $D(L)$ incident to L_i, we want the edge (L_i, L_{cand}) that makes the smallest angle with the base edge when measured counter-clockwise from (L_i, R_i) [file:1].
- **Right candidate** R_{cand}:
  - *Definition:* The first point encountered when moving **clockwise** from R_i around the right convex hull that forms a valid potential cross edge with L_i [file:1].
  - *Geometric meaning:* Among all edges in $D(R)$ incident to R_i, we want the edge (R_i, R_{cand}) that makes the smallest angle with the base edge when measured clockwise from (R_i, L_i) [file:1].

Why these directions? Counter-clockwise from the left and clockwise from the right ensures we're moving *upward* geometrically, consistent with building cross edges from bottom to top [file:1].

Candidate Validation via Edge Deletion Before we can use these candidates, we must ensure they don't violate the Delaunay property [file:1]. This requires checking and potentially **deleting edges** from $D(L)$ and $D(R)$ [file:1].

Left candidate validation:

1. Let L_{cand} be the initial candidate from the previous step [file:1].
2. Let L_{next} be the next point counter-clockwise from L_{cand} around L_i (i.e., the next potential candidate) [file:1].
3. **InCircle test:** Check if L_{next} lies inside the circumcircle of triangle $(L_i, L_{\text{cand}}, R_i)$ [file:1].

For points A, B, C in counter-clockwise order, point D lies inside their circumcircle when this determinant is positive [file:1]:

$$\begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{vmatrix} > 0$$

4. **If yes** (InCircle returns *true*):
   - The edge (L_i, L_{cand}) violates the Delaunay property and must be **deleted** from $D(L)$ [file:1]
   - Update $L_{\text{cand}} = L_{\text{next}}$ [file:1]
   - Repeat from step 2 with the new L_{cand} [file:1]
5. **If no:** L_{cand} is a valid candidate. Stop [file:1].

Why this works: If L_{next} is inside the circumcircle of $(L_i, L_{\text{cand}}, R_i)$, then the triangle formed by potential cross edge (L_{cand}, R_i) violates the Delaunay property [file:1]. The edge (L_i, L_{cand}) must be removed, and we consider the next candidate [file:1]. We continue until we find a candidate whose circumcircle (with the base) doesn't contain the next candidate [file:1].

Right candidate validation: Perform the symmetric process [file:1]:

6. Let R_{cand} be the initial candidate [file:1].
7. Let R_{next} be the next point clockwise from R_{cand} around R_i [file:1].
8. Check if R_{next} lies inside the circumcircle of $(R_i, R_{\text{cand}}, L_i)$ [file:1].
9. If yes, delete (R_i, R_{cand}), update $R_{\text{cand}} = R_{\text{next}}$, and repeat [file:1].
10. If no, R_{cand} is valid [file:1].

Mathematical guarantee: After validation, both L_{cand} and R_{cand} (if they exist) are points such that adding a cross edge to them would create a triangle satisfying the local Delaunay property [file:1].

Selecting the Next Cross Edge Now we have two validated candidates: L_{cand} (from the left) and R_{cand} (from the right) [file:1].

The decision criterion [file:1]:

- **If only one candidate exists:** Use that one (the other side may have no valid candidate above the current base) [file:1].
- **If both candidates exist:** Perform the **InCircle test** on the quadrilateral $(L_i, L_{\text{cand}}, R_i, R_{\text{cand}})$ [file:1]:

- Test if R_{cand} lies inside the circumcircle of triangle $(L_i, L_{\text{cand}}, R_i)$ [file:1]
- **If yes:** Choose (L_i, R_{cand}) as the next cross edge. Set $R_{i+1} = R_{\text{cand}}$, $L_{i+1} = L_i$ [file:1].
- **If no:** Choose (L_{cand}, R_i) as the next cross edge. Set $L_{i+1} = L_{\text{cand}}$, $R_{i+1} = R_i$ [file:1].

Geometric intuition: We're asking: "If we added the left candidate's cross edge, would the right candidate violate it?" If yes, we must choose the right candidate instead [file:1]. The candidate we choose is the one whose circumcircle **excludes** the other candidate [file:1].

Mathematical correctness: This ensures that the new cross edge (L_{i+1}, R_{i+1}) forms a triangle (with the current base edge) that satisfies the empty circumcircle property [file:1]. No point from either triangulation lies inside this circumcircle [file:1].

Iteration and Termination

1. **Add the edge:** Insert the chosen cross edge (L_{i+1}, R_{i+1}) into the triangulation [file:1].
2. **Update the base:** Set the current base edge to (L_{i+1}, R_{i+1}) [file:1].
3. **Repeat:** Return to the candidate finding steps with the new base edge and continue adding cross edges [file:1].

The process terminates when we can no longer find any valid candidates on either side [file:1]. This happens when we've reached the **upper tangent** of the two convex hulls—the topmost edge connecting them [file:1].

2.5.4 Implementation Details

The implementation uses a **Quad-Edge Data Structure** as specified in the original Guibas-Stolfi paper [file:1].

Quad-Edge Structure Each `Edge` represents a **directed** edge org $\rightarrow$ dest [file:1].

Fields:

- `org`: starting point [file:1]
- `dest`: ending point [file:1]
- `sym`: the symmetric (reverse) edge [file:1]
- `onext`: the next CCW edge around `org` [file:1]
- `oprev`: the next CW edge around `org` [file:1]
- `data`: used as a deletion flag [file:1]

Rationale for Quad-Edge Structure The divide-and-conquer Delaunay algorithm relies heavily on topology, not just geometry [file:1]. During the merge step, the algorithm must [file:1]:

- Walk CW and CCW around hulls
- Traverse edges around a vertex in rotational order
- Insert new cross edges between triangulations
- Delete edges that violate the InCircle test
- Splice and unsplice entire edge-rings
- Access the symmetric (reversed) direction of an edge instantly

The quad-edge structure uniquely supports these operations in $O(1)$ **time** [file:1].

Operations like `onext`, `oprev`, `sym`, `splice()`, `connect()`, and `delete_edge()` directly correspond to the motions of the merge algorithm: moving along hulls, rotating around endpoints, stitching triangulations together, and enforcing the Delaunay condition [file:1].

Alternative representations (adjacency lists, half-edge meshes, etc.) cannot support these local rotations and splices efficiently [file:1]. They require maintaining sorted adjacency lists or rebuilding structure after each edge modification, making the merge step impossible to implement efficiently [file:1].

Therefore, the quad-edge structure is used because it provides the exact topological operations needed to implement the merge step cleanly and in linear time [file:1].

2.5.5 Time Complexity Analysis

Let $T(n)$ be the time to triangulate n points [file:1].

- **Sorting:** $O(n \log n)$ initially [file:1]
- **Recurrence [file:1]:**
  - Divide: $O(1)$ to find the median [file:1]
  - Conquer: $2T(n/2)$ for two recursive calls [file:1]
  - Merge: $O(n)$ — at most $O(n)$ edges are examined and potentially deleted [file:1]

The recurrence relation is [file:1]:

$$T(n) = 2T(n/2) + O(n)$$

By the Master Theorem, this gives $T(n) = O(n \log n)$ [file:1].

Merge Step Time Complexity (Detailed)

- **Edge examination bound:** Each edge in $D(L)$ or $D(R)$ can be examined at most once as a potential candidate [file:1]. When we delete an edge, it's never examined again [file:1].
- **Candidate searches:** Finding each candidate involves walking around convex hull edges, and each hull edge is traversed at most once during the entire merge [file:1].

Total work [file:1]:

- Initial base edge: $O(n)$
- Each cross edge addition: $O(1)$ InCircle tests plus $O(k)$ edge deletions, where k edges are deleted
- Total edge deletions across entire merge: $O(n)$ (each edge deleted at most once)
- Number of cross edges added: $O(n)$

Sum: $O(n) + O(n) + O(n) = O(n)$ [file:1]

$$\boxed{\textbf{Total Time Complexity: } O(n \log n)}$$

2.5.6 Space Complexity Analysis

- $O(n)$ for storing the triangulation [file:1]
- $O(\log n)$ for the recursion stack [file:1]

$$\boxed{\textbf{Total Space Complexity: } O(n)}$$

2.5.7 Proof of Correctness

The proof proceeds by induction on the number of points [file:1].

Theorem 4 *The divide-and-conquer algorithm correctly computes the Delaunay Triangulation of any finite point set in* $\mathbb{R}^2$.

[Proof Sketch] We argue by induction on the number of points [file:1].

Base case: For 2–3 points, the triangulation is trivially correct as verified by direct construction [file:1].

Inductive step: Assume the algorithm correctly triangulates sets of size $< n$. For a set of size n [file:1]:

1. By the inductive hypothesis, $D(L)$ and $D(R)$ are valid Delaunay triangulations [file:1].
2. During merging, we maintain the invariant that all existing triangles satisfy the Delaunay property [file:1].

3. When adding a new cross edge, the InCircle test ensures no point lies within the circumcircle of newly formed triangles [file:1].

4. Edge deletion removes only those edges whose adjacent triangles would violate the Delaunay property after adding the new cross edge [file:1].

5. The lower tangent property guarantees that all points in one half lie on the correct side of the merging boundary [file:1].

Therefore, the final triangulation satisfies the empty circumcircle property for all triangles [file:1].

3 Implementation Details

3.1 Programming Language & Libraries

- **Language Used:** Python 3.
- **Core Libraries:**
  - **NumPy** (v1.24.4): Essential for numerical operations, array manipulation (point storage), vector mathematics, and geometric primitives such as dot and cross products.
  - **Matplotlib** (v3.7.1): Used for visualizing triangulations, plotting edges, and performing empirical time complexity analysis.
  - **SciPy.spatial.Delaunay** (SciPy v1.10.1): Used as a reference implementation to verify the correctness of custom algorithms.
  - **Time and Random Modules:** Used for performance measurement and random point generation.

3.2 Design Choices

The implementation covers three primary algorithms: **Randomized Incremental Construction (RIC)**, **Bowyer-Watson (BW)**, and **Divide and Conquer (D&C)**.

- **Data Structures:**
  - **Quad-Edge Structure (D&C):** Manages connectivity of edges, vertices, and faces, enabling $\mathbf{O(1)}$ topological updates during the merge step.
  - **Directed Acyclic Graph (DAG, RIC):** Facilitates point location within triangles in expected $\mathbf{O}(\log \mathbf{n})$ time by maintaining a history of triangle splits.
  - **Half-Edge Map (BW, RIC, Edge-Flipping):** A dictionary mapping a directed edge $(\mathbf{a}, \mathbf{b})$ to its adjacent triangle or opposite vertex $\mathbf{c}$. Enables $\mathbf{O(1)}$ lookup for circumcircle checks and Lawson edge flips.

- **Handling Degenerate Cases:**
  - **Super-Triangle (Incremental Methods):** RIC and BW start with a large super-triangle encompassing all points. Triangles connected to super-triangle vertices are removed after insertion.
  - **Numerical Robustness:** Degeneracies such as co-linear or co-circular points were handled using standard 64-bit floating-point arithmetic with a small ϵ tolerance in determinant-based predicates like **is_in_circumcircle** and **orient**.
- **Spatial Partitioning Optimizations:**
  - **Point Location DAG (RIC):** Reduces the search for containing triangles from $\mathbf{O(n)}$ to expected $\mathbf{O(\log n)}$.
  - **Sorting (D&C):** Input points are sorted by x-coordinate (and y-coordinate as tie-breaker) to facilitate efficient spatial splitting.

3.3 Challenges Faced

- **Precision Errors:**
  - **Circumcircle Predicate:** Floating-point errors in determinant calculations could incorrectly classify non-Delaunay edges or trigger infinite edge flips. High-precision NumPy operations mitigated this, but full robustness would require arbitrary-precision libraries.
- **Topological Complexity (D&C):**
  - **Merge Procedure:** Correctly computing lower and upper common tangents and iteratively inserting cross edges while maintaining Quad-Edge links was challenging and error-prone.
- **Performance Bottlenecks:**
  - **Non-Optimized BW:** Without point location, the Bowyer-Watson algorithm exhibits $\mathbf{O(n^2)}$ average complexity, becoming slow for large datasets ($n > 10{,}000$).
  - **Recursive Overhead (D&C):** Python recursion introduced function call overhead, often offsetting the asymptotic $\mathbf{O(n \log n)}$ advantage compared to optimized library implementations.

4 Experimental Setup

4.1 Environment

All algorithms were implemented in **Python 3**. The experiments were conducted on a machine with the following hardware and software specifications:

- **Language:** Python 3.x.

The implementation relies on the following key libraries:

- `NumPy`: Used for high-performance array manipulations, vectorization of geometric calculations, and coordinate management.
- `SciPy` (`scipy.spatial.Delaunay`): Used as a highly optimized $\mathcal{O}(n \log n)$ "Ground Truth" benchmark to verify the correctness of our custom implementations.
- `Matplotlib`: Utilized for visualizing the resulting triangulations. Specifically, `FuncAnimation` and `LineCollection` were used to render the incremental progress of the algorithms.
- `Standard Libraries`:
  - `time`: For measuring wall-clock execution time.
  - `math`, `random`: For geometric predicates and random number generation.
  - `collections`: Utilized `defaultdict` and `deque` for managing adjacency lists and DAG structures efficiently.

5 Voronoi Diagrams and Voronoi Triangles

5.1 Definition and Motivation

Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in $\mathbb{R}^2$, the **Voronoi diagram** partitions the plane into regions $V(p_i)$ such that every point in $V(p_i)$ is closer to p_i than any other point in P.

Voronoi diagrams are widely used in computational geometry, spatial analysis, nearest-neighbor search, and mesh generation.

5.2 Relationship to Delaunay Triangulation

The Voronoi diagram is the geometric **dual** of the Delaunay triangulation:

- Each vertex of the Voronoi diagram corresponds to the circumcenter of a Delaunay triangle.
- Each edge of the Voronoi diagram is perpendicular to a Delaunay edge.
- Constructing a Voronoi diagram can be done by first computing the Delaunay triangulation and then connecting circumcenters of adjacent triangles.

5.3 Voronoi Triangles

A **Voronoi triangle** is formed by connecting the vertices (circumcenters of Delaunay triangles) that correspond to three mutually adjacent Delaunay triangles. These triangles are useful for:

- Mesh refinement and smoothing
- Interpolation of scattered data
- Visualization of spatial relationships

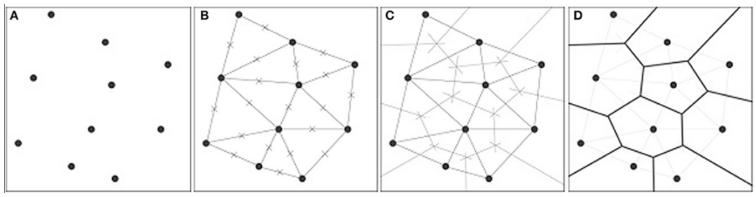


Figure 5: Construction of Voronoi diagrams.

5.3.1 Algorithm for Constructing Voronoi Diagram via Delaunay Triangulation

1. Compute the Delaunay triangulation of the point set P.
2. For each Delaunay triangle, compute its circumcenter.
3. Connect circumcenters of adjacent triangles to form Voronoi edges.
4. Assemble all Voronoi edges to form Voronoi cells.

5.3.2 Time and Space Complexity

- **Time Complexity:** $O(n \log n)$, dominated by the Delaunay triangulation step.
- **Space Complexity:** $O(n)$, as only circumcenters and adjacency information need to be stored.

5.4 Applications

Voronoi diagrams and triangles are applied in:

- Nearest-neighbor queries
- Mesh generation and finite element methods
- Geographical mapping and resource allocation
- Procedural generation in computer graphics

6 Results and Analysis

6.0.1 Metrics Used

The empirical analysis evaluates five Delaunay Triangulation algorithms—Divide and Conquer (DnC), Randomized Incremental Construction (RIC), SweepHull, Edge Flipping (EF), and Bowyer–Watson (BW)—across four point distributions (Uniform, Gaussian, Clustered, and Poisson Disc). Three core metrics were used:

- **Runtime (Wall-Clock Time):** Primary metric to evaluate computational efficiency and asymptotic scalability.
- **Memory Usage:** Measured in Megabytes (MB), used to compare space overhead of each algorithm.
- **Solution Quality:** Verified visually to ensure all algorithms produce a correct and topologically identical Delaunay Triangulation.

6.1 Divide and Conquer (DnC) Algorithm

6.1.1 Visual Validation and Solution Quality

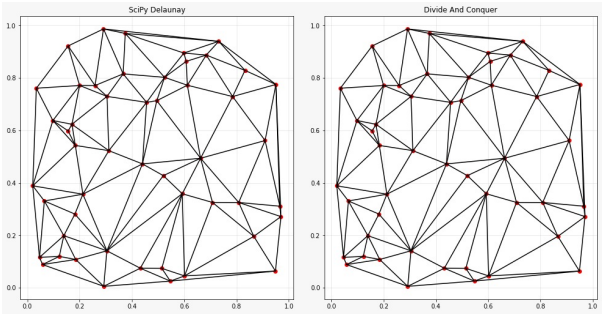


Figure 6: DnC on Uniform Distribution

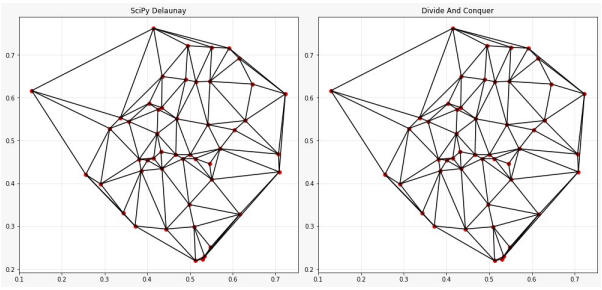


Figure 7: DnC on Gaussian Distribution

6.1.2 Runtime Analysis

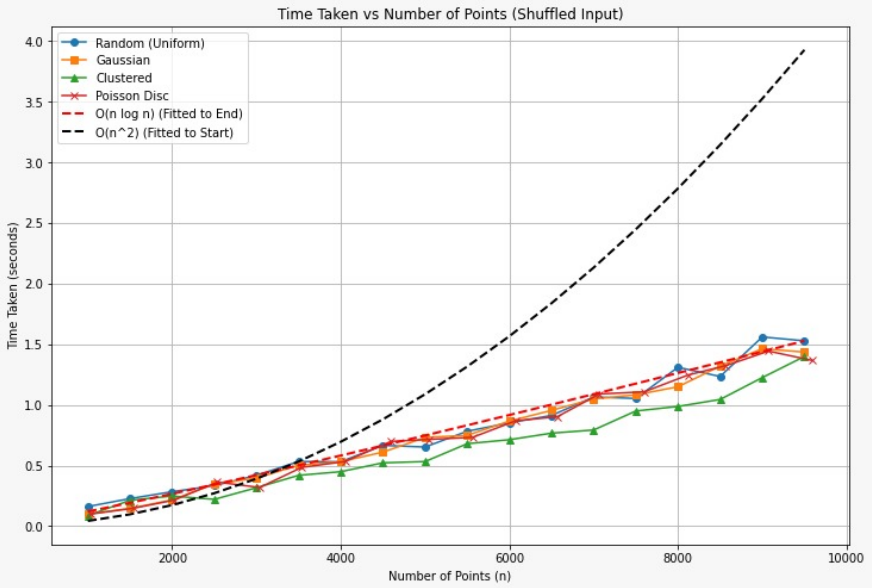
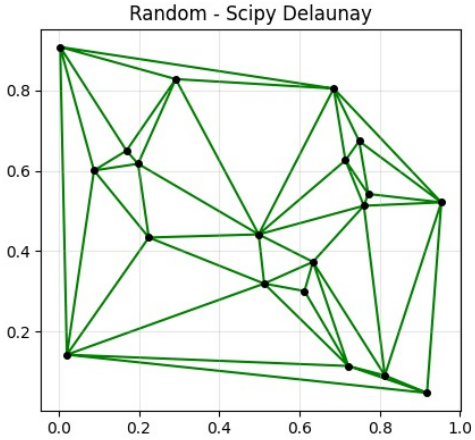


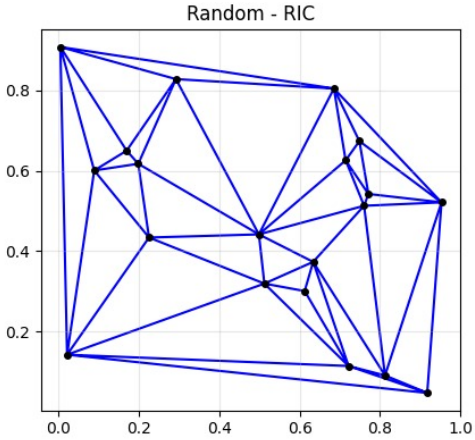
Figure 8: DnC Asymptotic Runtime following the theoretical $\mathbf{O}(\mathbf{n}\log\mathbf{n})$ curve.

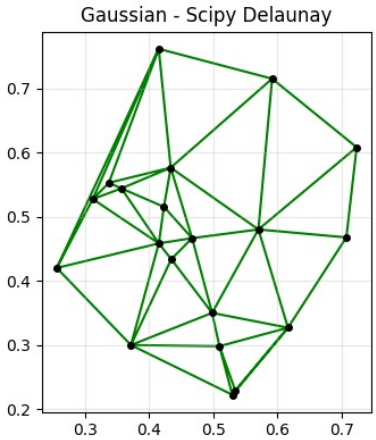
- **Complexity:** Matches the theoretical $\mathbf{O}(\mathbf{n}\log\mathbf{n})$ bound.
- **Data Sensitivity:** Minor variation across distributions; **Clustered** is fastest and **Gaussian** slightly slower due to uneven point spread.

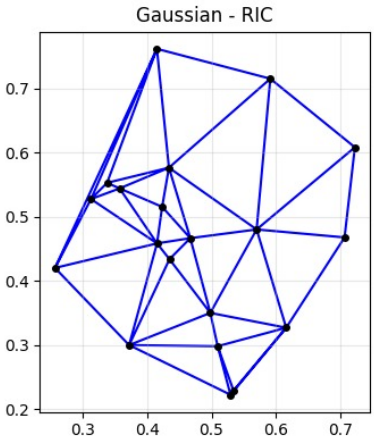
6.2 Randomized Incremental Construction (RIC)

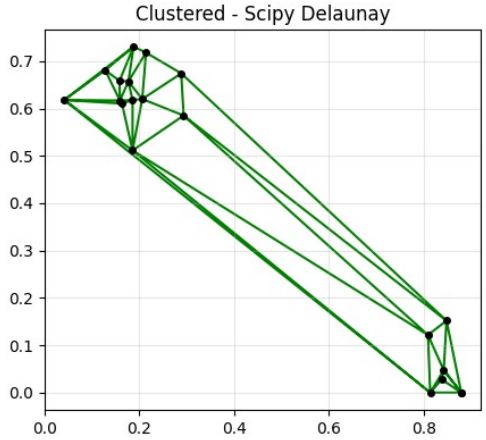
6.2.1 Visual Validation

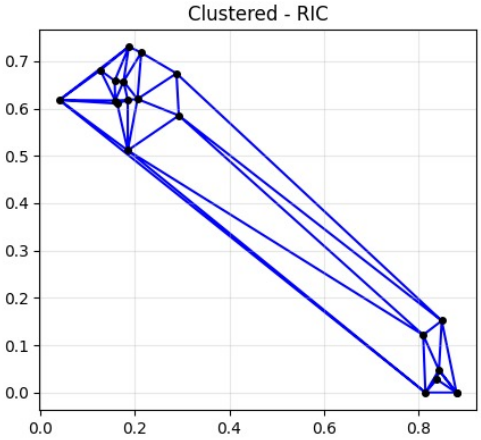


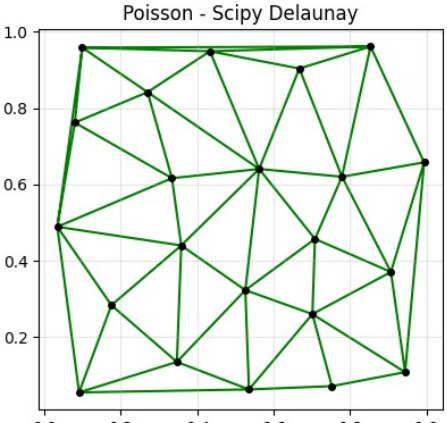


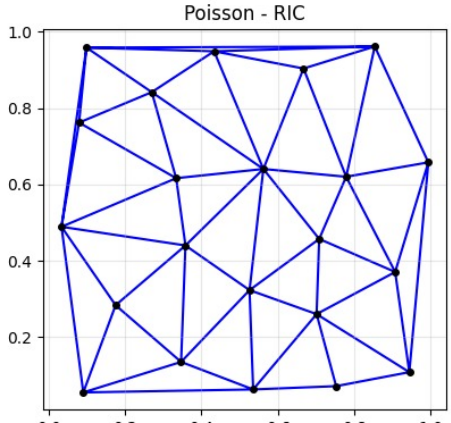












6.2.2 Runtime Analysis

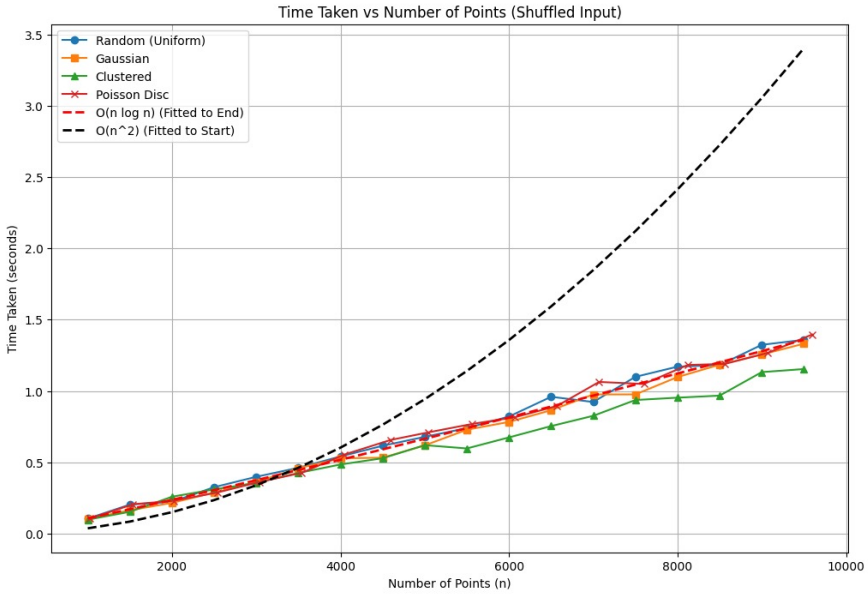
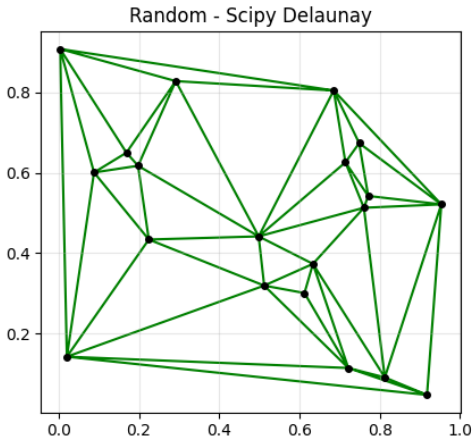


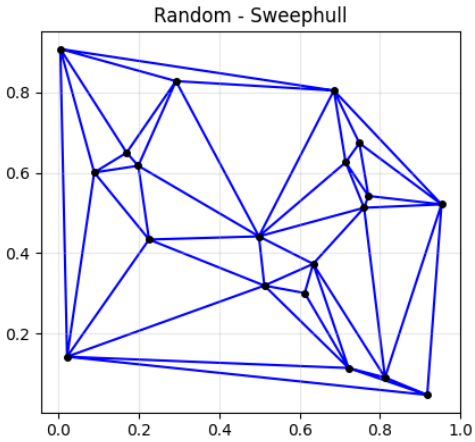
Figure 10: RIC Runtime following $\mathbf{O(n \log n)}$ asymptotics.

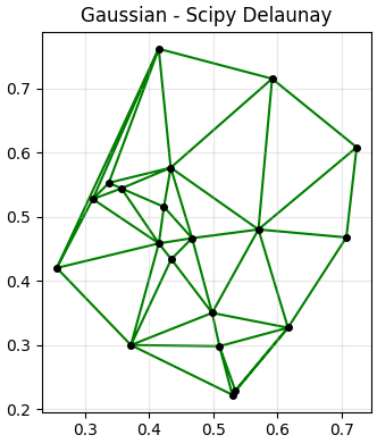
- **Complexity:** Empirically scales as $\mathbf{O(n \log n)}$.
- **Robustness:** Shows the **least variance** across distributions, due to random insertion order shielding the algorithm from pathological point sequences.

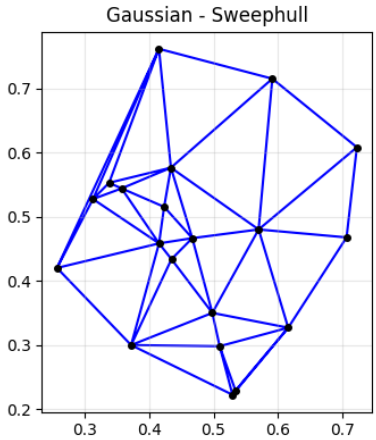
6.3 SweepHull Algorithm

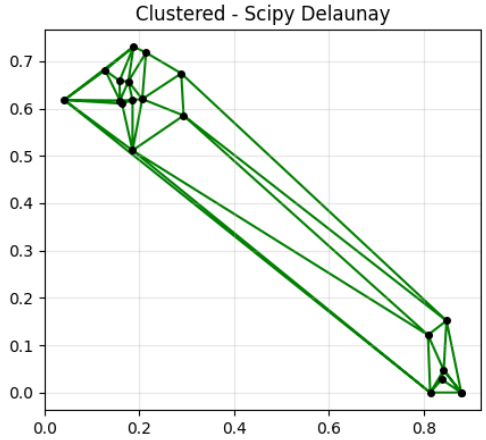
6.3.1 Visual Validation

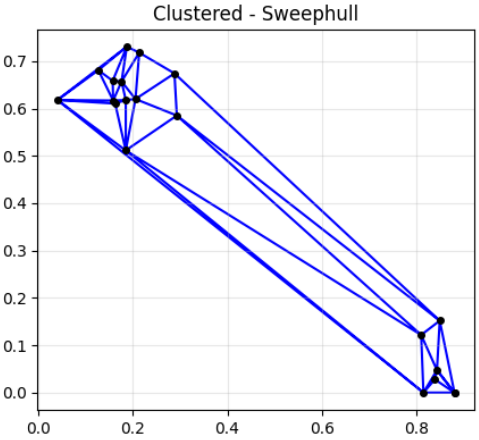


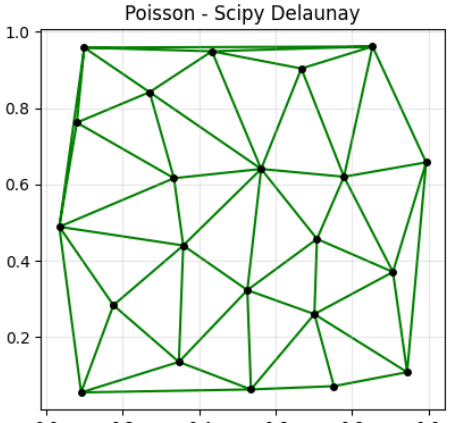


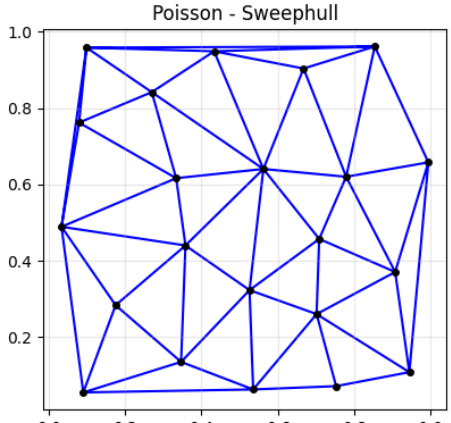












6.3.2 Runtime and Memory Analysis

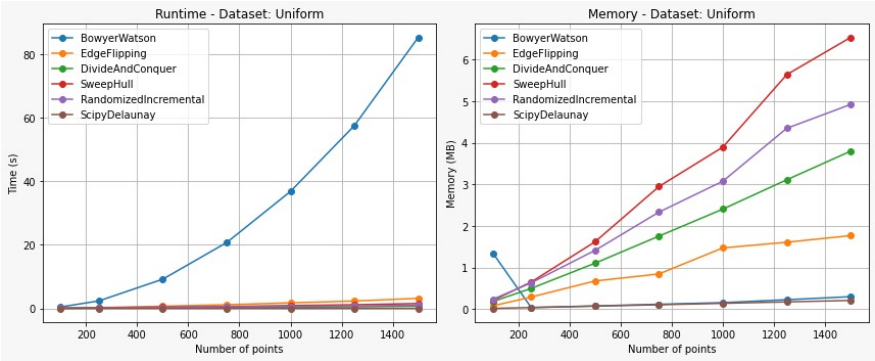
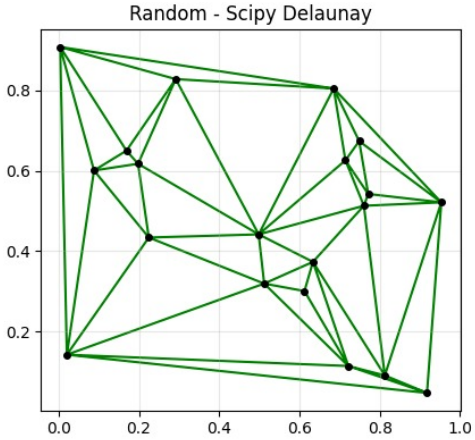


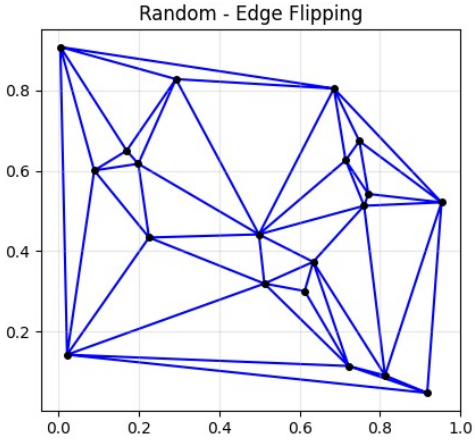
Figure 12: SweepHull Runtime (Left) and Memory (Right) on Uniform Dataset.

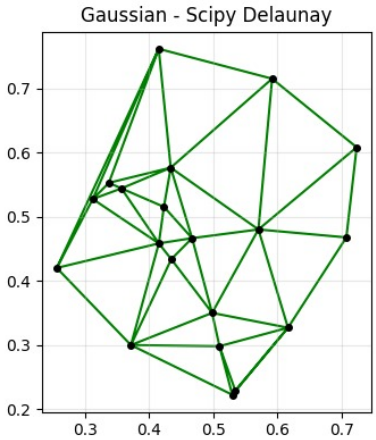
- **Complexity:** Follows $\mathbf{O(n \log n)}$, consistent with sweep-based hull expansion.
- **Memory Usage:** Highest memory footprint among optimal algorithms due to maintaining an elaborate sweep-line status structure.

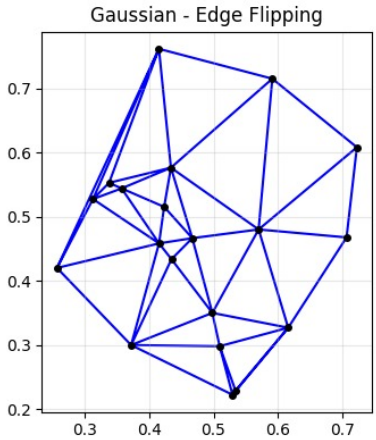
6.4 Edge Flipping (EF)

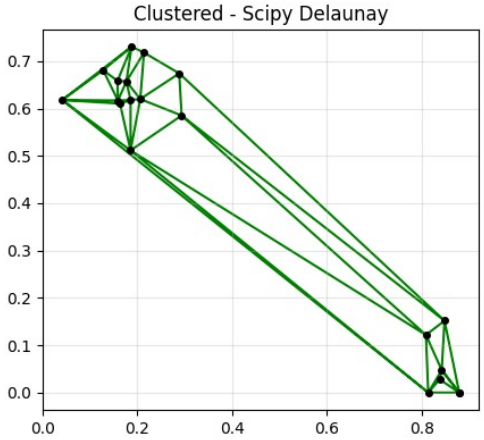
6.4.1 Visual Validation

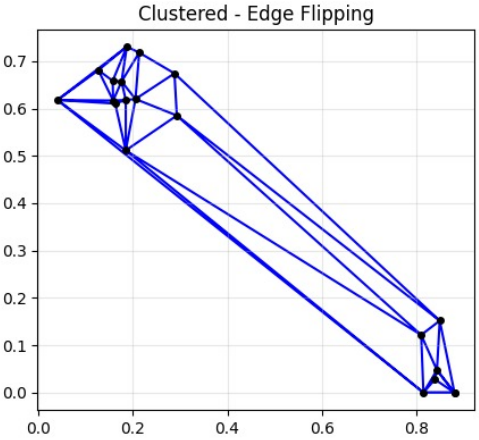


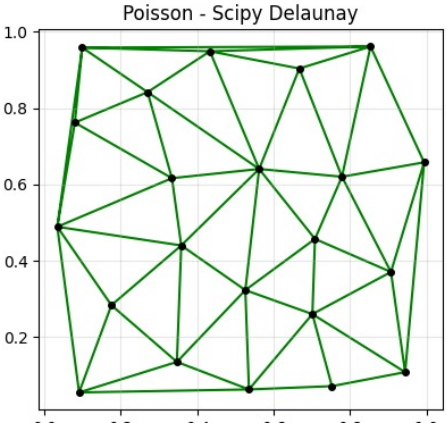


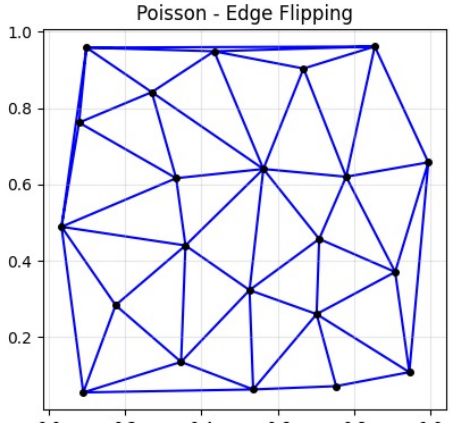












6.4.2 Runtime Analysis

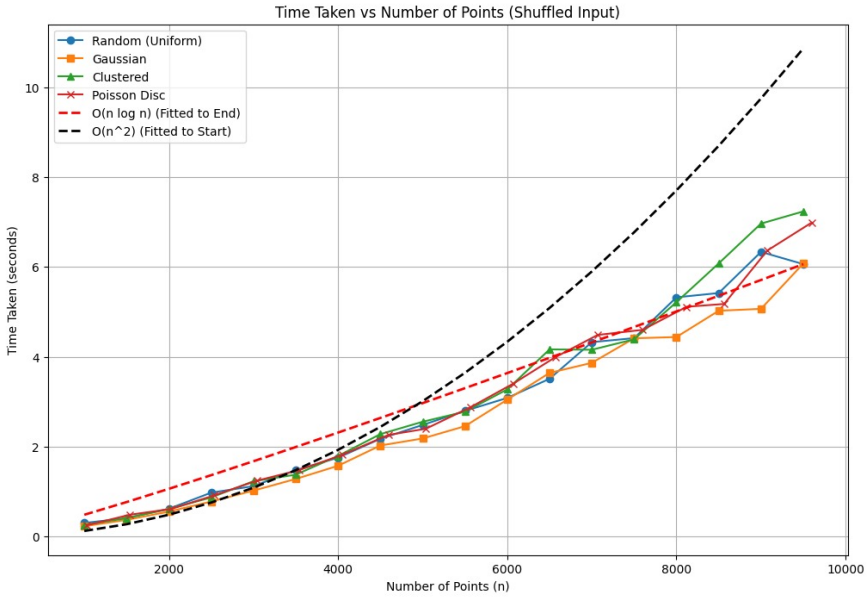
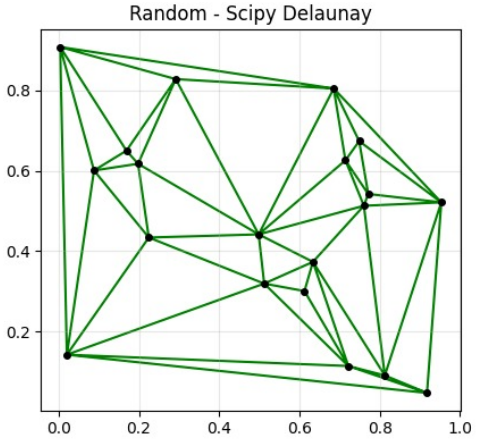


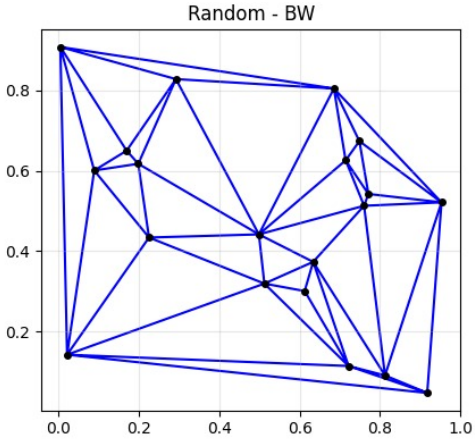
Figure 14: EF Runtime showing growth beyond $\mathbf{O(n \log n)}$.

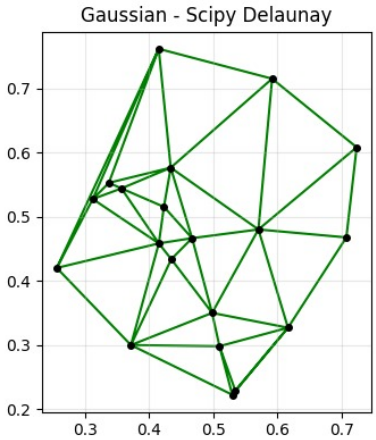
- **Complexity:** Empirical scaling transitions from $\mathbf{O(n \log n)}$ toward $\mathbf{O(n^{1.5})}$ as dataset size increases.
- **Sensitivity:** Performs worst on **Clustered** data due to elongated triangles requiring large numbers of iterative flips.

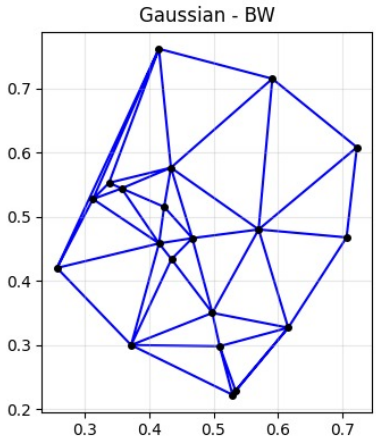
6.5 Bowyer–Watson (BW)

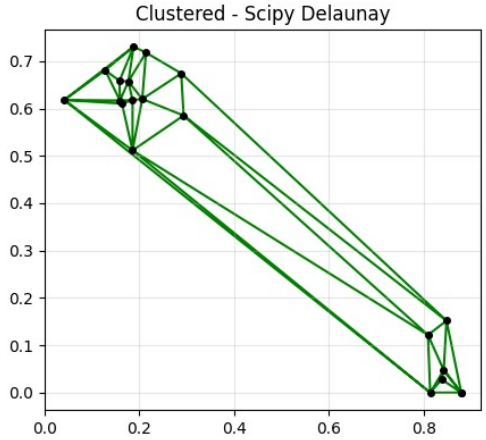
6.5.1 Visual Validation

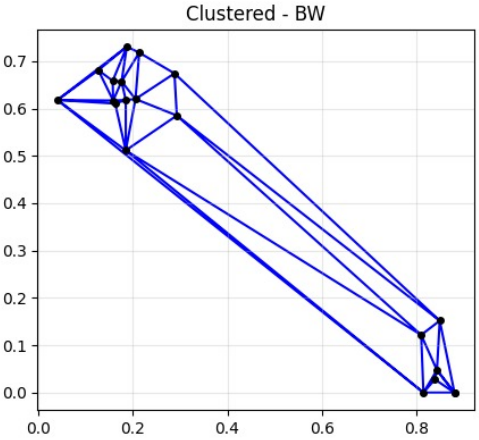


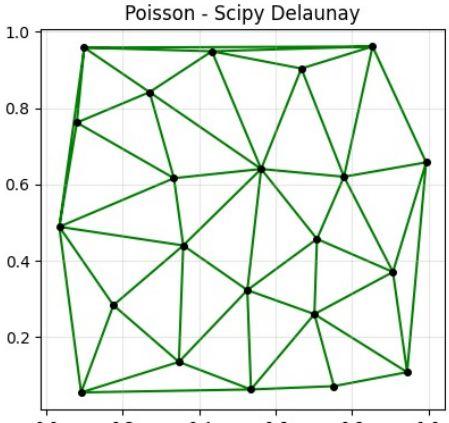


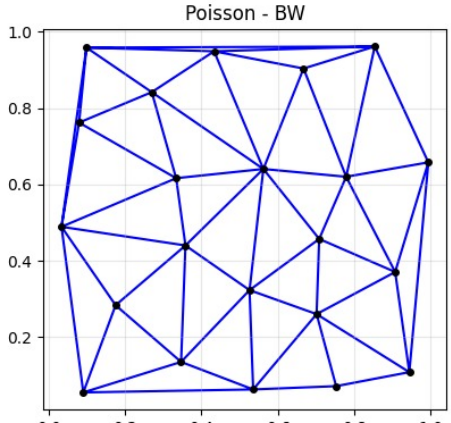












6.5.2 Runtime Analysis

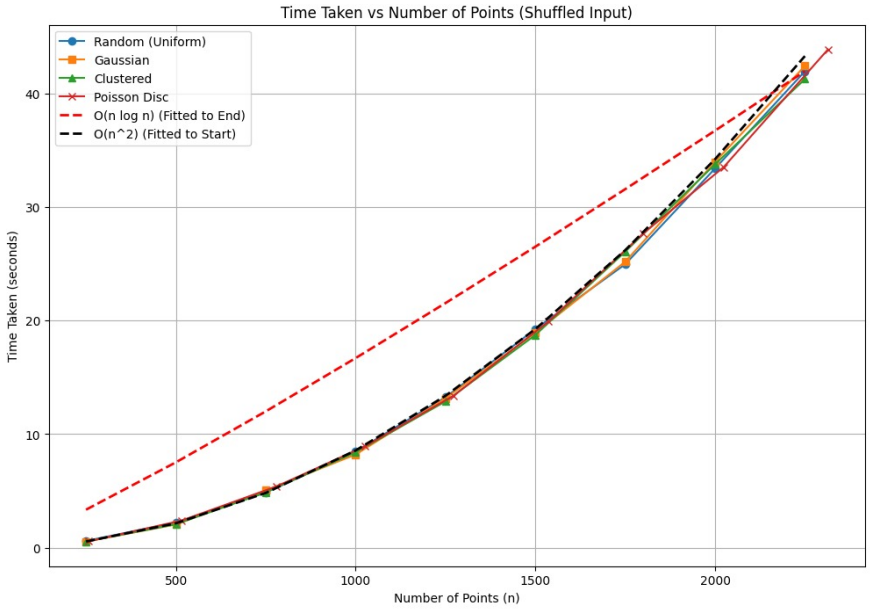


Figure 16: BW Runtime following $\mathbf{O(n^2)}$ behavior.

- **Complexity:** Confirms quadratic $\mathbf{O(n^2)}$ scaling due to repeated cavity reconstruction.

6.6 Comparative Summary

6.6.1 Runtime and Memory Comparison

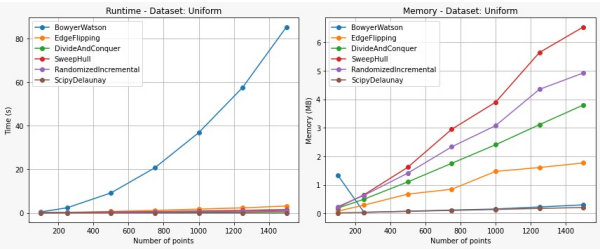


Figure 17: Runtime and Memory for Uniform Dataset

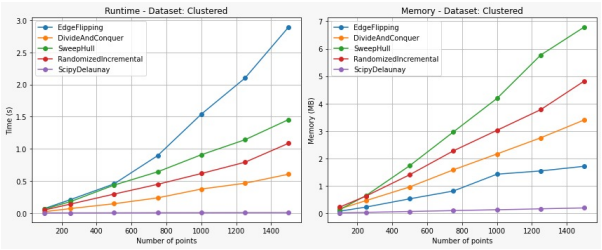


Figure 18: Runtime and Memory for Clustered Dataset

Algorithm	Theoretical	Empirical	Time (s)	Memory (MB)
DnC	$O(n \log n)$	$O(n \log n)$	≈ 0.6	≈ 3.8
RIC	$O(n \log n)$ (expected)	$O(n \log n)$	≈ 1.0	≈ 5.0
SweepHull	$O(n \log n)$	$O(n \log n)$	≈ 1.5	≈ 6.5
EF	$O(n \log n) + O(F)$	$O(n^{1.5})$	≈ 3.3	≈ 1.8
BW	$O(n^2)$ (worst)	$O(n^2)$	≈ 85.0	≈ 0.3

Table 1: Performance summary on Uniform dataset ($n = 1500$).

6.6.2 Interpretation of Results

The algorithms form three performance tiers:

- **Tier 1: Optimal ($O(n \log n)$) — DnC, RIC, SweepHull**
  - **DnC**: Fastest overall, best memory efficiency.
  - **RIC**: Most robust across distributions; randomness prevents worst-case behavior.
  - **SweepHull**: Matches optimal runtime but uses the most memory.
- **Tier 2: Sub-Optimal — Edge Flipping**
  - Runtime degrades toward $O(n^{1.5})$, though memory usage is moderate.
- **Tier 3: Non-Scalable — Bowyer–Watson**
  - Quadratic scaling makes BW unsuitable for datasets above a few thousand points.

7 Conclusion

7.1 Summary of Findings

The empirical analysis provided a comprehensive performance comparison of five Delaunay Triangulation algorithms—Divide and Conquer (DnC), Randomized Incremental Construction (RIC), SweepHull, Edge Flipping (EF), and Bowyer-Watson (BW)—against their theoretical complexities and against each other.

- **Asymptotic Complexity is Key:** The study showed that the $\mathbf{O(n \log n)}$ algorithms (DnC, RIC, SweepHull) are the only viable choices for scalable applications. The performance gap between these and the $\mathbf{O(n^2)}$ BW implementation becomes orders of magnitude for $n \geq 1500$.
- **DnC is the Performance Leader:** The **Divide and Conquer (DnC)** algorithm demonstrated the best overall performance, exhibiting the lowest runtime and highest memory efficiency among the $O(n \log n)$ group. Its speed is attributed to an efficient $O(n)$ merge step and a low constant factor.

- **RIC is the Robustness Leader:** The **Randomized Incremental Construction (RIC)** algorithm proved to be the most robust, showing minimal variance in runtime across all point distributions (Uniform, Gaussian, Clustered, Poisson Disc). This stability arises from the initial randomization step that regularizes insertion order.
- **Algorithm Degradation:** The **Edge Flipping (EF)** algorithm exhibited runtime degradation, deviating from its expected $\mathbf{O(n \log n)}$ trend toward approximately $\mathbf{O(n^{1.5})}$. This confirms its sensitivity to the number of flips (F), which grows rapidly for large or clustered inputs.
- **Worst Performer:** The **Bowyer-Watson (BW)** algorithm consistently showed a clear $\mathbf{O(n^2)}$ runtime trend, rendering it impractical for moderately sized or large point sets.

7.2 Limitations of the Current Implementation

While the analysis yielded clear comparative results, several limitations were observed:

- **Suboptimal Data Structures:** The poor $\mathbf{O(n^2)}$ performance of BW and the degradation of EF suggest that the underlying point-location and neighborhood-search structures were not optimal. For example, linear search or naïve lists were used instead of more advanced spatial structures such as a history graph, DAG, or bounding-box hierarchy.
- **Limited Memory Profiling:** Memory usage was only measured at the process level, without attributing consumption to specific internal structures (e.g., triangle list, edge list, locator structure). This prevented precise identification of why RIC and SweepHull incurred larger memory overheads.
- **Focus on 2D:** The analysis was restricted entirely to two-dimensional point sets. Extending these techniques to three dimensions (3D Delaunay Tetrahedralization) significantly increases computational complexity and introduces new challenges in data structure management.
- **Lack of Operation Counts:** The study did not log detailed operation counts (e.g., number of flips in EF, number of circumcircle tests in BW). Such quantitative metrics would strengthen the explanation for the observed runtime trends and provide finer diagnostic insight.

7.3 Future Work and Potential Improvements

Building on the limitations and findings, several promising directions for future work are identified:

- **Improved Data Structures:** Re-implement BW and RIC using optimal point-location structures (e.g., a *history graph*, *k-d tree*, or *Delaunay hierarchy*). This is expected to reduce BW's practical runtime toward $O(n \log n)$ and verify the theoretical $O(n \log n)$ behavior of RIC.

- **Parallelization of DnC and EF:** Investigate parallel variants of DnC and EF. The merge step in DnC and the independent flip operations in EF are highly parallelizable and may achieve near $O(\log n)$ effective runtime on multicore or GPU architectures.
- **Detailed Memory Profiling:** Perform structure-level memory profiling using tools such as Valgrind/Massif to isolate the memory-heavy components in SweepHull and RIC, enabling targeted optimizations.
- **3D Extension:** Extend the analysis to 3D Delaunay Tetrahedralization. Comparing algorithmic behavior in higher dimensions would provide valuable insight, as the complexity, robustness, and performance characteristics differ significantly from the 2D case.

8 Bonus Disclosure

- For each algorithm, we implemented a comparison of how the algorithm runs for different datasets, in particular - random dataset, gaussian dataset, poisson disc dataset and scattered dataset.
- We discussed Voronoi triangles and their dual relationship with Delaunay triangulation, highlighting how they help visualize spatial relationships in different datasets.

9 References

References

[1] C. L. Lawson, "Software for C^1 Surface Interpolation," in *Mathematical Software III*, J. R. Rice (Ed.), Academic Press, 1977, pp. 161–194.

[2] Leonidas Guibas and George Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics*, 4(2):74–123, April 1985. `https://doi.org/10.1145/282918.282923`.

[3] CP-Algorithms. Delaunay triangulation. `https://cp-algorithms.com/geometry/delaunay.html`.

[4] Samson's Math Site. Delaunay triangulation. `https://sites.google.com/site/samsoninfinite/multivariable-calculus/delaunay-triangulation`.