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 **4**
1.1 Problem Definition 4
1.2 Real-world Relevance 4
1.3 Objectives 5

2 Algorithm Descriptions **5**
2.1 Flipping Algorithm 5
2.1.1 Algorithm Introduction: Local Edge Flips 5
2.1.2 Empty Circumcircle Property and Flipping 5
2.1.3 Orientation Test 8
2.1.4 In-Circle Test 9
2.1.5 Time Complexity Analysis 9
2.1.6 Space Complexity Analysis 10
2.1.7 Proof Sketch of Correctness 11
2.2 Sweep Hull 11
2.2.1 Algorithm Introduction: Explanation and Intuition 11
2.2.2 Time Complexity Analysis 14
2.2.3 Space Complexity Analysis 14
2.2.4 Formal Proof of Correctness 14
2.3 Bowyer–Watson Algorithm 15
2.3.1 Algorithm Introduction: Incremental Delaunay via Cavity Retriangulation 15
2.3.2 Key Geometric Predicates 17
2.3.3 Time Complexity 18
2.3.4 Space Complexity 18
2.3.5 Proof Sketch of Correctness 19
2.4 Randomized Incremental Construction (RIC) 19
2.4.1 Algorithm Phases 19
2.4.2 Pseudocode 21
2.4.3 Asymptotic Analysis 21
2.4.4 Correctness and the Role of Randomization 22
2.5 Divide and Conquer Algorithm 23
2.5.1 Problem Statement 23
2.5.2 Algorithm Introduction and Intuition 23
2.5.3 Merge Procedure 24
2.5.4 Pseudocode 24
2.5.5 Time Complexity Analysis 24
2.5.6 Space Complexity 25
2.5.7 Proof Sketch of Correctness 25

3 Implementation Details **25**
3.1 Programming Language & Libraries . . . 25
3.2 Design Choices . . . 25
3.3 Challenges Faced . . . 26

4 Experimental Setup **26**
4.1 Environment . . . 26

5 Results and Analysis **26**
5.0.1 Metrics Used . . . 26
5.1 Empirical Results . . . 27
5.2 Comparison with Theory . . . 27
5.3 Observations and Interpretation . . . 27

6 Conclusion **28**

7 Bonus Disclosure **28**

8 References **28**

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.

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**.

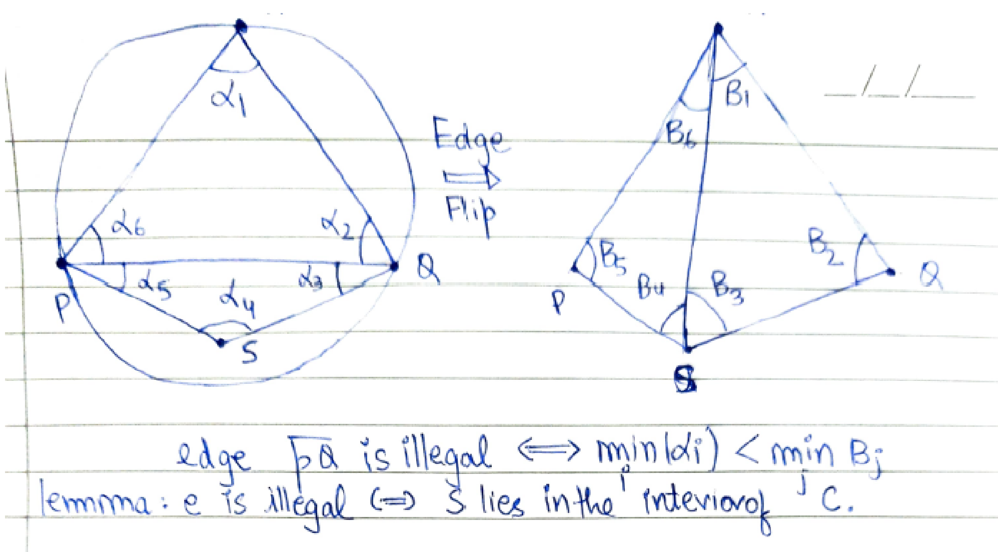
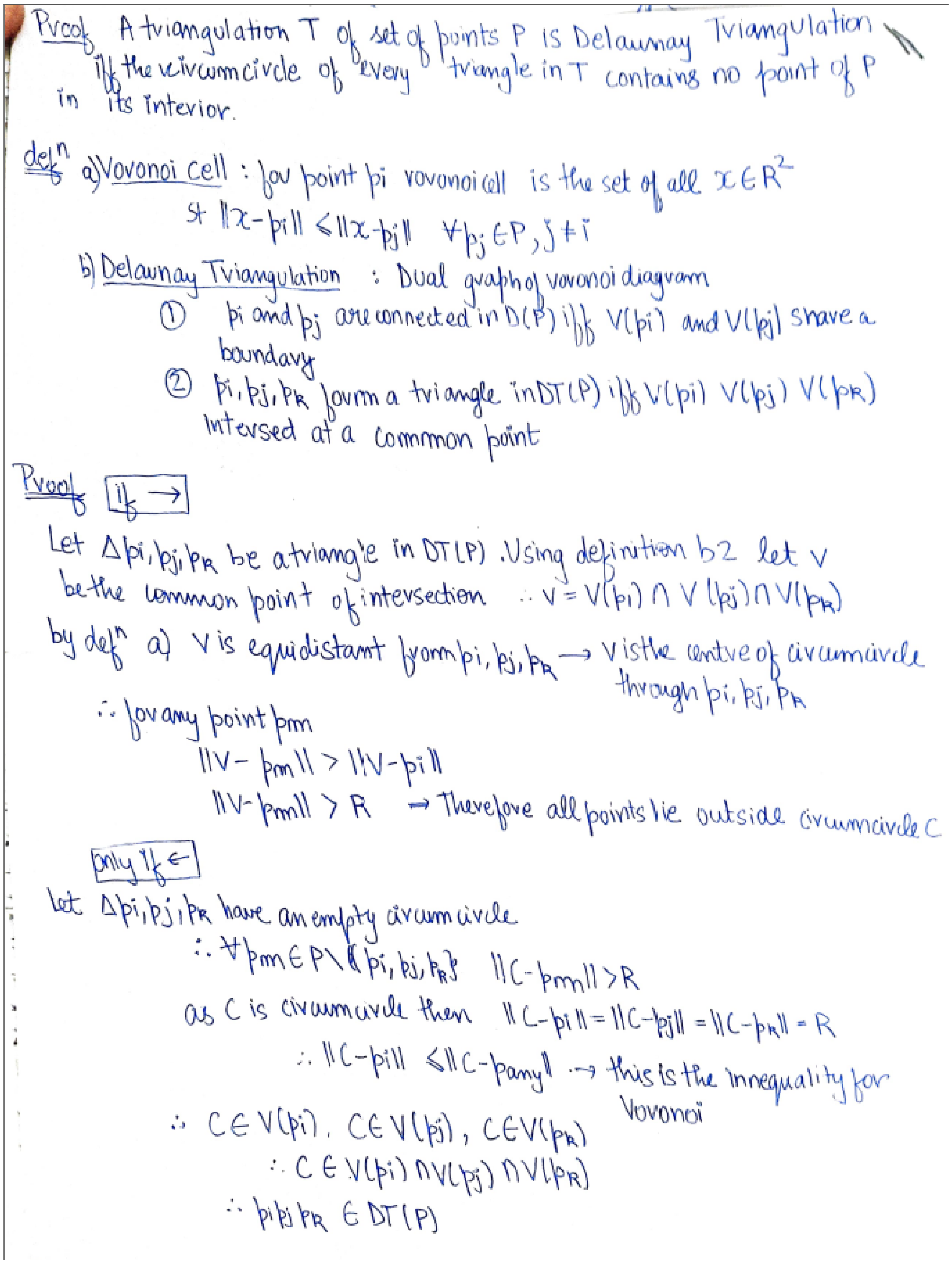


Figure 1



Proof A triangulation T of set of points P is Delaunay Triangulation iff the circumcircle of every triangle in T contains no point of P in its interior.

defn a) Voronoi cell : for point p_i voronoi cell is the set of all $x \in R^2$ st $\|x - p_i\| \leq \|x - p_j\| \ \forall p_j \in P, j \neq i$

b) Delaunay Triangulation : Dual graph of voronoi diagram

① p_i and p_j are connected in $D(P)$ iff $V(p_i)$ and $V(p_j)$ share a boundary

② p_i, p_j, p_k form a triangle in $DT(P)$ iff $V(p_i)$ $V(p_j)$ $V(p_k)$ intersect at a common point

Proof $\boxed{\text{if} \rightarrow}$

Let $\Delta p_i, p_j, p_k$ be a triangle in $DT(P)$. Using definition b2 let v be the common point of intersection $\therefore v = V(p_i) \cap V(p_j) \cap V(p_k)$

by defn a) v is equidistant from $p_i, p_j, p_k \rightarrow$ v is the centre of circumcircle through p_i, p_j, p_k

$\therefore$ for any point p_m

$\|v - p_m\| > \|v - p_i\|$

$\|v - p_m\| > R$ $\rightarrow$ Therefore all points lie outside circumcircle C

$\boxed{\text{only if} \leftarrow}$

let $\Delta p_i, p_j, p_k$ have an empty circumcircle

$\therefore \forall p_m \in P \setminus \{p_i, p_j, p_k\}$ $\|C - p_m\| > R$

as C is circumcircle then $\|C - p_i\| = \|C - p_j\| = \|C - p_k\| = R$

$\therefore \|C - p_i\| \leq \|C - p_{any}\|$ $\rightarrow$ this is the inequality for Voronoi

$\therefore C \in V(p_i)$, $C \in V(p_j)$, $C \in V(p_k)$

$\therefore C \in V(p_i) \cap V(p_j) \cap V(p_k)$

$\therefore p_i p_j p_k \in DT(P)$

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.

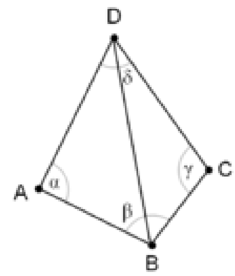


Figure 3

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

Circle equation

$(X-U)^2 + (Y-V)^2 = R^2$

where U and V $\Rightarrow$ (U,V) is circumcentre

R is radius

now a point D lies inside circumcircle when

$|D-O|^2 < R^2$

if we map a point $P(x,y)$ to $P'(x,y,x^2+y^2)$ in 3D space by lifting map to paraboloid $z = x^2+y^2$

3D points X', Y', Z', D' are coplanar if D in circumcircle of XYZ

Hence the sign of determinant

$$\begin{vmatrix} X_x & X_y & X_x^2+X_y^2 & 1 \\ Y_x & Y_y & Y_x^2+Y_y^2 & 1 \\ Z_x & Z_y & Z_x^2+Z_y^2 & 1 \\ D_x & D_y & D_x^2+D_y^2 & 1 \end{vmatrix}$$

determines if D is above or below the plane through X', Y', Z'

Hence on ABC clockwise sign +/- show D inside / outside the plane respectively if det = 0 on the circumcircle

Now simplify determinant by translating coordinates so D is at origin

$$\begin{bmatrix} X_x - D_x & X_y - D_y & (X_x - D_x)^2 + (X_y - D_y)^2 \\ Y_x - D_x & Y_y - D_y & (Y_x - D_x)^2 + (Y_y - D_y)^2 \\ Z_x - D_x & Z_y - D_y & (Z_x - D_x)^2 + (Z_y - D_y)^2 \end{bmatrix}$$

$\therefore$ Rule $\underline{\det > 0}$

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      |
                +-------------+--------------+
                              |
                              v
                +----------------------------+
                |  Find nearest neighbour xj |
                +-------------+--------------+
                              |
                              v
                +----------------------------+
                |  Choose xk forming smallest |
                |  circumcircle with x0, xj   |
                +-------------+--------------+
                              |
                              v
         +-----------------------------------------+
         | Compute circumcenter of seed triangle   |
         +-------------------+---------------------+
                             |
                             v
         +-----------------------------------------+
         | Radially sort remaining points around   |
         | the circumcenter                        |
         +-------------------+---------------------+
                             |
                             v
    +---------------------------------------------------+
    | Sweep-Hull Construction (Stage I):                |
    | Insert points in radial order, find visible hull  |
    | edges, form new triangles, update hull            |
    +------------------+--------------------------------+
                       |
                       v
    +---------------------------------------------------+
    | Delaunay Enforcement (Stage II):                  |
    | Apply in-circumcircle tests on adjacent           |
    | triangles; flip violating edges                   |
    +------------------+--------------------------------+
                       |
                       v
                +-----------------------------+
                | 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: $\Delta \to \{T_1, T_2, T_3\}$
    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 $\to \{T_a, T_b\}$
        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: $\mathbf{O}(\mathbf{n} \log \mathbf{n})$.

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:

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

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: $\mathbf{O}(\mathbf{n})$.

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 Algorithm

2.5.1 Problem Statement

We aim to perform **Delaunay Triangulation** on a set of points.

Formally, given a finite set of points

$$P = \{p_1, p_2, ..., p_n\} \subset \mathbb{R}^2,$$

the **Delaunay Triangulation** DT(P) is a triangulation where no point in P lies inside the circumcircle of any triangle in DT(P).

2.5.2 Algorithm Introduction and Intuition

The **Divide and Conquer (D&C) algorithm** is a recursive strategy to construct the Delaunay Triangulation efficiently, achieving an expected time complexity of $O(n \log n)$. The core idea is simple:

1. **Divide:** Split the point set P into two roughly equal subsets along a vertical line (or by x-coordinate).
2. **Conquer:** Recursively compute the Delaunay Triangulation for each subset.
3. **Merge:** Merge the two triangulations into a single Delaunay Triangulation by finding the *base edge* and performing local edge flips to restore the Delaunay property along the interface.

2.5.3 Merge Procedure

Merging is the most critical step:

- Identify the **lower common tangent** connecting the left and right triangulations.
- Incrementally add triangles by connecting points across the tangent while ensuring that no triangles violate the *Empty Circumcircle Property*.
- Perform edge flips locally as needed to maintain the Delaunay condition.
- Continue until all points between the left and right subsets are connected.

Figure 5: Illustration of the divide-and-conquer merge process.

2.5.4 Pseudocode

```
DivideAndConquerDT(P):
Input : Set of points P in R^2
Output: Delaunay Triangulation DT(P)

1. If |P| <= 3:
       Construct DT(P) directly
       Return DT(P)
2. Sort P by x-coordinate
3. Split P into PL and PR
4. TL := DivideAndConquerDT(PL)
5. TR := DivideAndConquerDT(PR)
6. Merge TL and TR into T using lower/upper common tangents
7. Return T
```

2.5.5 Time Complexity Analysis

- **Divide:** Sorting takes $O(n \log n)$, splitting is $O(1)$.
- **Conquer:** Each recursive call handles roughly half the points.
- **Merge:** Merging two triangulations can be done in linear time $O(n)$ using the tangent-based procedure.

The recurrence relation is:

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

By the Master Theorem:

$$\boxed{T(n) = O(n \log n)}$$

2.5.6 Space Complexity

- Storing points: $O(n)$
- Recursive stack: $O(\log n)$
- Triangulation edges and adjacency: $O(n)$

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

2.5.7 Proof Sketch of Correctness

1. **Base Case:** For $|P| \leq 3$, direct triangulation is trivially Delaunay.
2. **Inductive Step:** Assume the left and right triangulations are correct.
3. **Merge Correctness:** The merge procedure ensures that the lower and upper tangents connect the triangulations while performing all necessary edge flips. Locally enforcing the Empty Circumcircle Property guarantees global Delaunay correctness.
4. **Termination:** Recursion terminates when subsets reduce to size 3, and the merge is finite.

Therefore, the Divide and Conquer algorithm produces a correct Delaunay Triangulation of the original point set P in $O(n \log n)$ expected time.

3 Implementation Details

//figure out what to write

3.1 Programming Language & Libraries

List:

- Language used (Python, C++, etc.)
- Library versions (NumPy, Matplotlib, etc.)

3.2 Design Choices

Discuss:

- Data structures used (e.g., half-edge structure, adjacency lists)
- Handling of degenerate cases
- Spatial partitioning optimizations (if any)

3.3 Challenges Faced

Mention issues such as:

- Precision errors
- Input ordering effects
- Performance bottlenecks

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 Results and Analysis

// results of comparitive analysis of algos -¿ fill in after that is done

5.0.1 Metrics Used

- Runtime (wall-clock time)
- Memory usage
- Number of operations (comparisons, flips, etc.)

5.1 Empirical Results

Add your graphs/tables here.

`runtime_results.png`

Figure 6: Runtime comparison across datasets.

Algorithm	Time (ms)	Memory (MB)	Notes
SweepHull	X	Y	-
Bowyer–Watson	X	Y	-
Incremental	X	Y	-
RIC	X	Y	-

Table 1: Performance metrics summary.

5.2 Comparison with Theory

Discuss similarities/differences between measured and theoretical runtimes.

5.3 Observations and Interpretation

Discuss why the observed results occurred.

6 Conclusion

Summarize:

- Main findings
- Best-performing algorithm and why
- Limitations of your implementation
- Possible improvements (parallelization, better data structures)

7 Bonus Disclosure

Explicitly list:

- 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.

8 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.