# Voronoi Diagram

Chen Pery, Dolev Nissan

## 1 Introduction

Voronoi diagrams are a powerful and widely used tool in computational geometry. They partition a plane into regions based on the distance to a specific set of points, known as generators. Each region, called a Voronoi cell, contains all points closer to its generator than to any other point.

### Voronoi patterns in nature

## 2 Historical Background

The concept is named after the Russian-Ukrainian mathematician Georgy Voronoi, who studied, developed, and generalized this idea in 1908. However, the idea of dividing space into regions based on distance can be traced back to earlier work by René Descartes and Peter Gustav Lejeune Dirichlet. Voronoi extended Dirichlet's ideas and applied them to higher dimensions.

### 2.1 Dirichlet Tessellations(1850)

**Johann Peter Gustav Lejeune Dirichlet** (1805–1859), a known German mathematician, made significant contributions to number theory, analysis, and mathematical physics. One of his important contributions was the concept of what is now known as **Dirichlet Tessellations**.

Dirichlet was interested in the way space could be divided based on proximity to a set of points. Specifically, he was studying the arrangement of lattice points and how space could be partitioned into regions, each associated with the nearest lattice point. These regions, now called *Dirichlet regions*, were defined as follows:

$$T_i = \{x \in \mathbb{R}^n : \|x - p_i\| \leq \|x - p_j\| \text{ for all } j \neq i\}$$

Here, $p_i$ represents a point in the lattice, and $T_i$ is the region containing all points $x$ that are closer to $p_i$ than to any other point $p_j$ in the lattice. This partitioning of space into convex regions is the essence of Dirichlet Tessellations.

### 2.2 Voronoi Diagram (1908)

**Georgy Voronoi** (1868–1908) was a Russian and Ukrainian mathematician best known for his work in the field of number theory and for the development of what is now called the Voronoi diagram. Voronoi's work extended the ideas of Dirichlet to more general and higher-dimensional spaces, providing a framework that is widely used in computational geometry today.

The **Voronoi diagram** partitions a space into regions based on the distance to a set of given points. Each point in this set, called a "site" or "seed," has a corresponding region consisting of all the locations closer to it than to any other site. These regions are known as *Voronoi cells*. The formal definition of a Voronoi cell $V(p_i)$ for a site $p_i$ is:

$$V(p_i) = \{x \in \mathbb{R}^n : \|x - p_i\| \leq \|x - p_j\| \text{ for all } j \neq i\}$$

## 2.3 Uses of Voronoi Diagrams in Different Periods

### 2.3.1 19th Century to Early 20th Century

**Number Theory and Lattice Structures:** In the mid-19th century, Peter Gustav Lejeune Dirichlet utilized what we now recognize as Voronoi diagrams (Dirichlet tessellations) to study positive definite quadratic forms and the distribution of algebraic numbers. These early applications were primarily theoretical, focusing on partitioning n-dimensional space relative to a lattice of points.
**Crystallography Foundations:** Although not explicitly using Voronoi diagrams, the partitioning concepts influenced early crystallography studies, where the arrangement of atoms in a crystal lattice was of interest.

### 2.3.2 Mid 20th Century

**Geography and Meteorology:** In the 1950s and 1960s, Voronoi diagrams began to find applications outside of theoretical contexts, particularly in geography and meteorology. They were used to model areas of influence, such as the range of weather stations or the division of land for various ecological studies. For example, in meteorology, the Thiessen polygon method (a type of Voronoi diagram) was employed to estimate rainfall in different regions based on data from scattered weather stations. This allowed for more accurate predictions and analyses of weather patterns.

### 2.3.3 Late 20th Century

**Computers and Applied Fields:** With the advent of computers in the latter half of the 20th century, Voronoi diagrams became a powerful tool for solving practical problems across various applied fields. In robotics, Voronoi diagrams were used for motion planning, allowing robots to navigate by finding paths that maximize the distance from obstacles. In computer graphics, they were employed to generate procedural textures and terrains. Additionally, Voronoi diagrams became integral to Geographic Information Systems (GIS) for tasks such as spatial analysis and resource allocation. For instance, urban planners used Voronoi diagrams to determine the most efficient placement of services like schools and hospitals. Moreover, Voronoi diagrams have been employed to map the large-scale structure of the universe, dividing space into regions around galaxies, clusters, and other celestial bodies. This helps in modeling galaxy distributions and understanding cosmic structures.

### 2.3.4 21st Century

**Ubiquiti Across Disciplines:** In the 21st century, Voronoi diagrams have become ubiquitous across many disciplines. In computer science, they are a key component of clustering algorithms like k-means, where the space is divided into clusters based on proximity to a set of centroids. In biology, Voronoi diagrams are used to model the growth of cells and the structure of tissues, providing insights into processes like tumor growth and tissue engineering. Urban planning continues to benefit from Voronoi diagrams, particularly in the design and optimization of layouts for infrastructure like cell towers and emergency services. Their ability to model and optimize spatial relationships makes them invaluable in modern technology and science.

# 3 Definitions and Properties

## 3.1 Definition of Voronoi Diagrams

Given a set of distinct points (also called *sites* or *generators*) in the Euclidean plane:

$$\{P_1, P_2, \ldots, P_k\} \subset \mathbb{R}^2,$$

the Voronoi diagram of this set is a partition of the plane $\mathbb{R}^2$ into regions such that each point $P_i$ corresponds to a region $V(P_i)$. This region consists of all points in the plane that are closer to $P_i$ than to any other site $P_j$. Formally, the Voronoi cell $V(P_i)$ associated with site $P_i$ is defined as:

$$V(P_i) = \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; d\left((x, y), P_i\right) < d\left((x, y), P_j\right),\ \forall j \neq i \right\},$$

where $d$ denotes the Euclidean distance:

$$d\left((x, y), P_i\right) = \sqrt{(x - x_i)^2 + (y - y_i)^2}.$$

The Voronoi diagram $\mathcal{V}$ is the collection of all Voronoi cells:

$$\mathcal{V} = \bigcup_{i=1}^{k} V(P_i).$$

Alternatively, the Voronoi diagram can be viewed as the complement of the union of all Voronoi cells in $\mathbb{R}^2$:

$$\mathbb{R}^2 \setminus \bigcup_{i=1}^{k} V(P_i),$$

which consists of the Voronoi edges and vertices where the distance to two or more sites is equal.

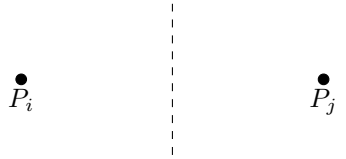
### Understanding Voronoi Cells as Intersections of Half-Planes

For any two distinct sites $P_i$ and $P_j$, the set of points in $\mathbb{R}^2$ that are closer to $P_i$ than to $P_j$ is defined by:

$$H_{ij} = \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; d\left((x, y), P_i\right) < d\left((x, y), P_j\right) \right\}.$$

This set $H_{ij}$ is a half-plane bounded by the perpendicular bisector of the line segment $\overline{P_i P_j}$. The perpendicular bisector is the set of points equidistant from $P_i$ and $P_j$.

**Visualization:**




- *Half-Plane* $H_{ij}$: One side of the plane divided by the perpendicular bisector of $\overline{P_i P_j}$.
- *Perpendicular Bisector*: The line where $d\left((x, y), P_i\right) = d\left((x, y), P_j\right)$.

### Constructing Voronoi Cells

The Voronoi cell $V(P_i)$ can be constructed by taking the intersection of all half-planes $H_{ij}$ for $j \neq i$:

$$V(P_i) = \bigcap_{\substack{j=1 \\ j \neq i}}^{k} H_{ij} = \bigcap_{\substack{j=1 \\ j \neq i}}^{k} \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; d\left((x, y), P_i\right) < d\left((x, y), P_j\right) \right\}.$$

Since each $H_{ij}$ is an open convex set (a half-plane), their intersection $V(P_i)$ is also an open convex set. This means that each Voronoi cell is:

- **Convex**: Any line segment connecting two points within $V(P_i)$ lies entirely within $V(P_i)$.
- **Possibly Unbounded**: Voronoi cells at the boundary of the set of sites may extend infinitely.

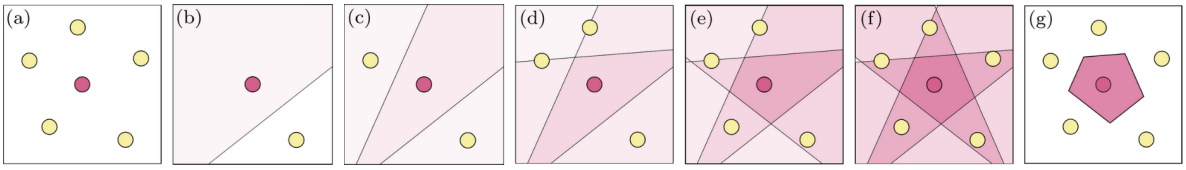



Figure 1: (a) A central magenta particle and five neighboring yellow particles; (b)–(f) construction of the Voronoi cell of the central particle, shown as a series of intersections of half-spaces, where each is defined by the central particle and one of its neighbors. (g) The completed Voronoi cell of the central particle.

## 3.2 Properties of Voronoi Diagrams

Consider a set of $k$ distinct points $\{P_1, P_2, \ldots, P_k\}$ in the Euclidean plane $\mathbb{R}^2$, and let $V(P_i)$ denote the Voronoi cell corresponding to the site $P_i$. The **Voronoi diagram** $\mathcal{V}$ of this set is defined as:

$$\mathcal{V} = \mathbb{R}^2 \setminus \bigcup_{i=1}^{k} V(P_i)$$

The Voronoi diagram $\mathcal{V}$ consists of edges (finite or infinite line segments) and vertices (points where edges meet) that partition the plane based on the proximity to the sites. The edges and vertices of the Voronoi diagram possess several important properties, which are detailed below.

1. **Edge Equidistance:** Any point $Q$ located on a Voronoi edge between cells $V(P_i)$ and $V(P_j)$ is equidistant to exactly two sites $P_i$ and $P_j$. Formally:

   $$d(Q, P_i) = d(Q, P_j) = r$$

   and for all other sites $P_l$ where $l \neq i, j$:

   $$d(Q, P_l) > r, \quad \forall l \neq i, j$$

   Thus, the circle centered at $Q$ with radius $r$ passes through $P_i$ and $P_j$ and contains no other sites from the set $\{P_1, P_2, \ldots, P_k\}$ within its interior.

2. **Vertex Equidistance:** A Voronoi vertex $W$, where three or more Voronoi edges converge, is equidistant to three (or possibly more) sites $P_i$, $P_j$, and $P_l$:

   $$d(W, P_i) = d(W, P_j) = d(W, P_l) = r > 0$$

   and for all other sites $P_m$ where $m \neq i, j, l$:

   $$d(W, P_m) > r, \quad \forall m \neq i, j, l$$

   Thus, the circle centered at $W$ with radius $r$ passes through $P_i$, $P_j$, and $P_l$, containing no other sites from the set within its interior.

3. **Unbounded Cell:** The Voronoi cell of a point $P_m$ is infinite $\Leftrightarrow$ $P_m$ is on the boundary of the convex hull of the set $\{P_1, P_2, ..., P_k\}$.
   Indeed, consider all the segments $\{P_m, P_i\}, i = 1, 2, ..., m-1, m+1, ..., k$. $P_m$ is on the convex hull boundary $\Leftrightarrow$ the vectors $P_i - P_m$ lie in a half-plane (i.e., there is a direction $\vec{n}$ such that $\langle (Pi - Pm) \cdot \vec{n} \rangle > 0, \forall i \neq m$). If this is the case, we'll have that all points on the ray $P_m - t\vec{n}$ are closer to $P_m$ than any other points:

   $$d[(P_m + t\vec{n}, P_m)]^2 = \langle (P_m + t\vec{n} - P_m)(P_m + t\vec{n} - P_m) \rangle = t^2 \langle \vec{n}, \vec{n} \rangle = t^2$$

   $$d[(P_m + t\vec{n}, P_i)]^2 = \langle (P_m + t\vec{n} - P_i)(P_m + t\vec{n} - P_i) \rangle = d[P_m, P_i]^2 + 2\langle (P_m - P_i), \vec{n}t \rangle + t^2 + 2 > t^2$$

   Therefore, if $P_m$ is on the convex boundary, then $V(P_m)$ is unbounded. The opposite is also easy to show by considering a point $P_s$ and showing that if $P_s$ is inside the convex hull of the points, i.e., $P_s$ is such that the vectors $P_i - P_s$ do not lie in a half-plane, then we can select three of them so that the corresponding half-planes intersect to form a finite triangle.

4. **Convexity of Voronoi Cells:** Each Voronoi cell $V(P_i)$ is a convex polygon (possibly unbounded). This convexity arises because $V(P_i)$ is the intersection of half-planes:

$$V(P_i) = \bigcap_{\substack{j=1 \\ j \neq i}}^{k} H_{ij},$$

where each $H_{ij}$ is a half-plane defined by:

$$H_{ij} = \left\{ (x, y) \in \mathbb{R}^2 \;\middle|\; d((x, y), P_i) \leq d((x, y), P_j) \right\}.$$

Since the intersection of convex sets is convex, $V(P_i)$ is convex.

5. **Complexity Bounds:** Assuming the sites are in general position (no three sites are colinear and no four are cocircular), the Voronoi diagram for $k$ sites has the following bounds:

$$V \leq 2k - 5, \quad E \leq 3k - 6.$$

(a) **Modeling as a Planar Graph:**
The Voronoi diagram is a planar graph where:

- *Vertices* correspond to Voronoi vertices.
- *Edges* correspond to Voronoi edges.
- *Faces* correspond to Voronoi cells.

(b) **Euler's Formula for Planar Graphs:**
Euler's formula:

$$V - E + F = 2.$$

(c) **Counting Faces:**
The number of faces is equal to the number of sites:

$$F = k.$$

(d) **Degree of Vertices:**
Each Voronoi vertex has degree 3:

$$\sum \deg(v) = 3V = 2E \implies E = \frac{3V}{2}.$$

(e) **Applying Euler's Formula:**
Substitute $E = \frac{3V}{2}$ and $F = k$:

$$V - \frac{3V}{2} + k = 2 \implies -\frac{V}{2} + k = 2 \implies V = 2k - 4.$$

(f) **Calculating Edges:**
Using $E = \frac{3V}{2}$:

$$E = \frac{3}{2}(2k - 4) = 3k - 6.$$

Therefore:

$$V = 2k - 4, \quad E = 3k - 6.$$

The bounds are:

$$V \leq 2k - 5, \quad E \leq 3k - 6.$$

# 4 Delaunay Triangulation

Delaunay triangulation is a geometric structure that divides a set of points into non-overlapping triangles such that no point is inside the circumcircle of any triangle. In simpler terms, the Delaunay triangulation connects points in a way that maximizes the minimum angle of each triangle, avoiding skinny triangles and ensuring a stable and well-formed triangulation.

## 4.1 Properties of Delaunay Triangulation:

1. Maximizes Minimum Angles: Among all possible triangulations of a set of points, the Delaunay triangulation maximizes the minimum angle of the triangles, which helps avoid long, thin triangles.
2. Dual of Voronoi Diagram: The Delaunay triangulation is the dual graph of the Voronoi diagram. That means if you connect the centers of the circumcircles of adjacent triangles, you obtain the Voronoi diagram.
3. Circumcircle Property: No point in the triangulation is inside the circumcircle of any triangle. This is often referred to as the "empty circumcircle" property.
4. Efficient for Pathfinding: Delaunay triangulation is widely used in pathfinding, surface reconstruction, mesh generation, and 3D modeling because of its efficiency and stability.

## 4.2 Formal Definition

A **Delaunay triangulation** for a set of points

$$P = \{p_1, p_2, \ldots, p_k\}$$

in the plane $\mathbb{R}^2$ is a triangulation such that no point in $P$ lies inside the circumcircle of any triangle in the triangulation. Formally, for every triangle $\triangle p_i p_j p_l$ in the triangulation, the circumcircle $C$ passing through $p_i$, $p_j$, and $p_l$ satisfies:

$$C \cap P = \{p_i, p_j, p_l\}.$$

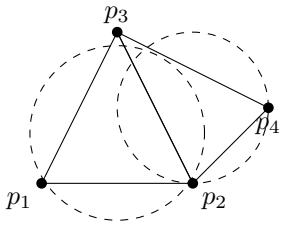



Figure 2: Delaunay Triangulation with Circumcircles

## 4.3 Construction of Delaunay Triangulation

The Delaunay triangulation can be constructed using several algorithms. One of the most common is the incremental algorithm, which works by adding one point at a time to an existing triangulation and maintaining the Delaunay condition.

1. **Initialization: Super-Triangle**

   We create a super-triangle that is large enough to encompass all the points in the input set $P = \{p_1, p_2, \ldots, p_k\}$. This super-triangle serves as a preliminary structure that ensures all input points are within a known boundary, facilitating the handling of boundary conditions.

2. **Add Points Incrementally**

   For each point $p_i \in P$, we identify the triangle in the current triangulation that contains $p_i$. then, we split this triangle into three smaller triangles by connecting $p_i$ to the vertices of the original triangle.

3. **Edge Flipping to Restore the Delaunay Condition**

   After the addition of a new point, the resulting triangulation may not satisfy the Delaunay condition. To restore the Delaunay property, we need to check if any newly created edges violate this condition. Specifically, we have to examine whether any point lies inside the circumcircle of any triangle. we can achieve this by performing *edge flipping*; An edge flip involves swapping the diagonal of the quadrilateral formed by two adjacent triangles, thereby restoring the Delaunay property. We repeat the edge-flipping process when necessary until all triangles satisfy the Delaunay condition.

4. **Remove the Super-Triangle** Once all points are added and the triangulation is adjusted, we remove the super-triangle and any associated triangles or edges that were part of it.

## 4.4 Delaunay Triangulation and Voronoi Diagram Duality

In a Delaunay triangulation, each triangular face has a corresponding circumcircle. Let $p_1, p_2, p_3$ denote the three vertices of a triangle in the Delaunay triangulation, and let $C$ represent its circumcircle. The center of $C$, which is the point equidistant from $p_1, p_2$, and $p_3$, corresponds to a vertex in the Voronoi diagram.

Once the vertices of the Voronoi diagram are determined, we can establish a mapping between the edges of the Delaunay triangulation and the edges of the Voronoi diagram. Specifically, an edge in the Voronoi diagram between two vertices is the perpendicular bisector of the corresponding edge in the Delaunay triangulation. This bisector separates the regions associated with the two seed points, and as such, each edge in the Voronoi diagram corresponds to an edge in the Delaunay triangulation, and vice versa.

Conversely, starting from a Voronoi diagram, we can construct the Delaunay triangulation. In this case, each seed point in the Voronoi diagram corresponds to a vertex in the Delaunay triangulation. An edge between two seed points in the Voronoi diagram will map to the perpendicular bisector of the corresponding edge in the Delaunay triangulation between these two vertices.

Thus, the Delaunay triangulation and the Voronoi diagram are dual structures: the vertices of the Delaunay triangulation correspond to the seed points of the Voronoi diagram, and the edges of one structure correspond to the bisectors of the edges in the other.

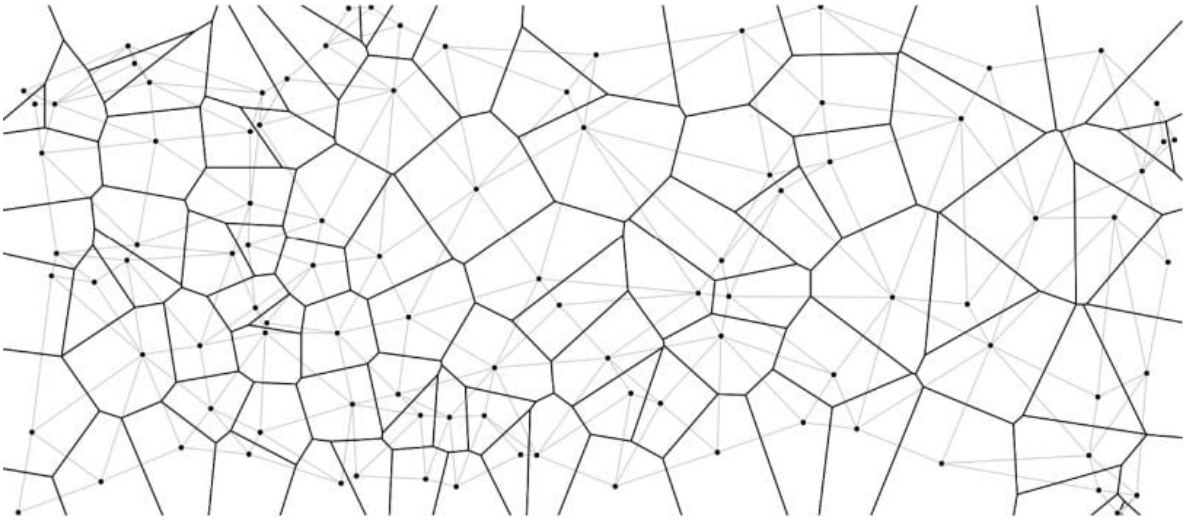


Figure 3: Voronoi diagram with Delaunay triangulation. — Image: Francesco Bellelli

# 5 Code Implementation of Voronoi Diagram

We implemented the Voronoi diagram using Python, leveraging the `scipy` package. The code was developed in a Colab notebook and is available in the file `Voronoi_diagram.ipynb`. Our implementation utilizes the `scipy.spatial.Voronoi` class, which internally employs the Qhull algorithm. The construction process involves the following steps:

## 5.1 Algorithm Steps

1. **Delaunay Triangulation:**

   Compute the Delaunay triangulation of the input points. This triangulation connects points to form triangles such that the circumcircles of the triangles contain no other points from the input set. The Delaunay triangulation serves as the foundation for constructing the Voronoi diagram.

2. **Dual Graph Construction:**

   The Voronoi diagram is the dual graph of the Delaunay triangulation. For each Delaunay triangle, a Voronoi vertex is placed at the circumcenter of the triangle. Voronoi edges are then drawn between these vertices, effectively constructing the Voronoi diagram.

3. **Voronoi Cell Creation:**

   Voronoi cells are formed by connecting the Voronoi vertices corresponding to adjacent Delaunay triangles. Each Voronoi cell represents the region closest to a specific input point, delineated by the boundaries formed through this dual relationship.

## 5.2 Implementation Details

- **Language:** Python
- **Libraries:** `scipy` for computational geometry, specifically `scipy.spatial.Voronoi`
- **Environment:** Google Colab notebook (`Voronoi_diagram.ipynb`)
- `scipy.spatial.Voronoi`: Utilized to compute the Voronoi diagram based on the input set of points. This function internally uses the Qhull algorithm to perform the necessary computations.

The Voronoi algorithm, implemented using the Qhull algorithm in `scipy`, requires a minimum of four non-collinear points in a two-dimensional space to form an initial simplex, which in 2D is a triangle composed of three points. Ensuring that no three points are collinear is crucial to prevent the collapse of Voronoi regions into undefined lines and to maintain clear, distinct boundaries between Voronoi cells. This requirement allows the algorithm to construct valid circumcircles necessary for determining Voronoi vertices, thereby ensuring that each Voronoi region is properly enclosed around its corresponding input point without ambiguity. Consequently, having at least four non-collinear points is essential for generating a valid and stable Voronoi diagram, effectively preventing scenarios where regions become overlapping or extend infinitely without clear demarcation, and ensuring that the boundaries between regions accurately represent the proximity relationships among the input points.

## 5.3 Complexity Analysis

The time complexity of constructing the Voronoi diagram using the Qhull algorithm is $O(n \log n)$ for two-dimensional data, where $n$ is the number of input points. The algorithm operates through the following phases:

- **Divide Step:**

  Qhull recursively divides the set of input points into smaller subsets, analogous to the divide step in merge sort. This division continues until each subset is small enough to be processed directly, typically containing pairs or triples of points. The divide step involves $O(\log n)$ recursive steps.

- **Conquer Step:**

  After solving the smaller subproblems (Voronoi cells in smaller regions), the algorithm merges these solutions to form the global Voronoi diagram. The merging involves stitching together adjacent regions, similar to how merge sort combines two sorted arrays. The merge step at each level requires $O(n)$ time.

Thus, the overall time complexity of the algorithm is $O(n \log n)$.

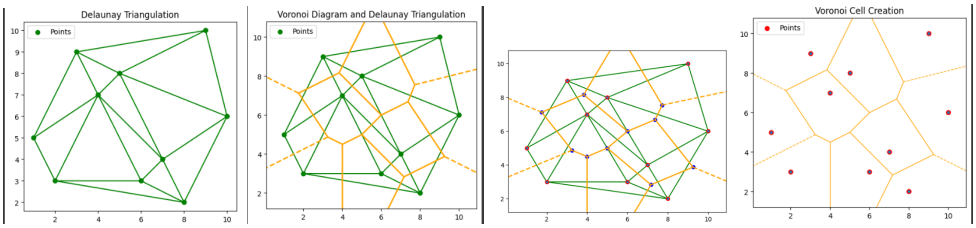



Figure 4: Results of running our code

### 5.4 Storage of the Voronoi Diagram

we should store the following :

- **Vertices:** The points where the bisectors intersect.
- **Edges:** Line segments that connect the vertices and form the boundaries between Voronoi cells.
- **Regions:** Each region (Voronoi cell) is defined by a point (site) and the edges (bisectors) that bound its area.

Most computational geometry libraries store Voronoi diagrams as a combination of these components. For example, Python's scipy.spatial.Voronoi class stores:

- **Vertices:** The coordinates of the intersection points of bisectors.
- **regions:** A list of indices into the vertices array defining each Voronoi cell's polygon.
- **ridge_vertices:** Indices of vertices that form the boundary (ridges) between two neighboring points.

## 6 Our Reaserch

At the beginning of our research, we explored the following key applications of Voronoi diagrams in robotics:

- **Multi-Agent Systems and Task Allocation :** In multi-agent systems, such as swarm robotics, the focus is on designing and coordinating large numbers of relatively simple robots. These robots collaborate to perform tasks that are difficult or impossible for a single robot to accomplish. Inspired by the behavior of biological swarms like ants, bees, and birds, swarm robotics leverages collective behavior principles to achieve coordination and task completion through local interactions among robots and with the environment. Voronoi diagrams play a role in allocating regions of responsibility to different robots within a multi-robot system. For example, in search-and-rescue operations, each robot can be assigned a Voronoi region to search. This ensures that robots do not overlap their search areas, preventing wasted resources and ensuring comprehensive coverage of the search area.
- **Path Planning:** Path planning is essential in robotics, involving the navigation of robots from a starting point to a destination while avoiding obstacles and optimizing criteria such as distance and safety. Voronoi diagrams provide a natural framework for path planning by generating paths

that maintain equidistance from obstacles, thereby enhancing safety. However, purely Voronoi-based paths may not always represent the shortest possible routes in terms of distance. To address this, hybrid approaches integrate Voronoi edges with shortest-path algorithms like A* or BFS, balancing safety with efficiency. In known environments, robots can precompute Voronoi cells and navigate along their edges to reach destinations safely. Conversely, in unknown or dynamic settings, robots dynamically generate and update Voronoi diagrams based on real-time sensor data. This allows them to adaptively navigate and avoid newly detected obstacles, ensuring that they can efficiently and safely traverse complex environments.

Building upon these applications, our research focuses on improving the algorithms for maintaining dynamic Voronoi diagrams, which are essential for real-time task allocation and path planning in multi-agent systems. By optimizing the update process, we aim to enhance the real-time responsiveness and scalability of multi-agent systems, enabling more robust and flexible task allocation and path planning in dynamic and large-scale environments.

# 7 Dynamic Voronoi Diagram

we include `Dynamic_Voronoi_diagram.py` in our submission, which updates the Voronoi diagram in real-time as points are added.

## 7.1 Algorithmic Approaches

Several algorithms have been developed to efficiently maintain dynamic Voronoi diagrams, each with varying computational complexities and implementation strategies. As detailed in Section 6.3 on Complexity Analysis, the **Divide-and-Conquer Algorithm** recursively splits the set of seed points into smaller subsets, constructs Voronoi diagrams for each subset independently, and then merges them to form the complete diagram. This approach has a computational complexity of $O(n \log n)$ for constructing the diagram from scratch [2]. However, due to the higher computational overhead during the merging phase, it is less optimal for highly dynamic scenarios where frequent updates are required.
Another prominent method is the **Incremental Algorithm**, introduced by Asano et al. (1991), which handles point insertions and deletions with an average-case complexity of $O(\log n)$ per update [1].

In these approaches, efficiently identifying neighboring points is important for minimizing the computational complexity of dynamic updates. Rapid neighbor identification reduces the scope of recalculations needed when points are added or removed, thereby enhancing the overall performance and scalability of Voronoi diagram maintenance. Spatial data structures such as **k-d trees** and **ball trees** can be used to achieve this efficiency, enabling neighbor searches in O(logn) time

- **k-d Trees:** In our case, 2-d trees are binary tree structures that efficiently organize points in a two-dimensional space to facilitate rapid nearest neighbor searches. The operation of a k-d tree begins with the root node, which splits the entire 2D space along the x-axis at the median x-coordinate of all points. This ensures a balanced partitioning, with approximately half of the points lying to the left and the other half to the right of the splitting line. The recursive partitioning process continues as the left child of the root node further splits its subset of points along the y-axis at the median y-coordinate, while the right child performs a similar split along the y-axis. The recursion persists until each leaf node contains a small number of points (typically one or two). The k-d trees offering an average-case time complexity of O(logn) for nearest neighbor searches.[6]

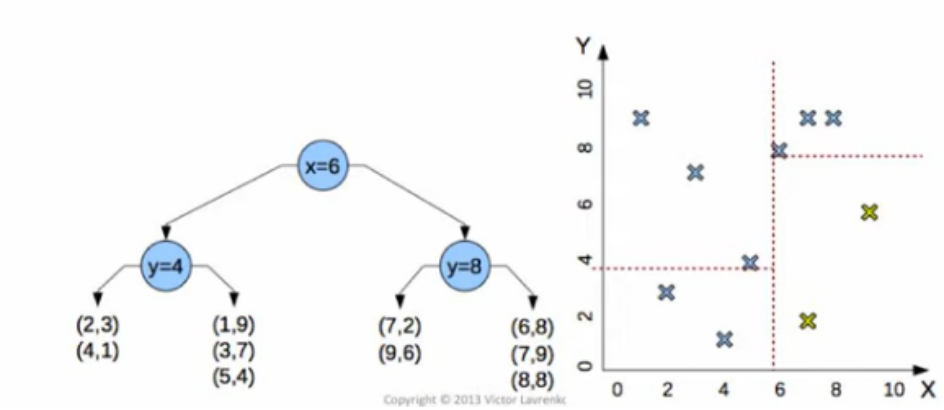



Figure 5: Illustration of a 2D k-d Tree Structure

- **Ball Trees:** Ball trees are another effective spatial data structure used for organizing points in 2D space, particularly beneficial for high-dimensional data but equally applicable in 2D. Unlike k-d trees, which partition space using hyperplanes, ball trees encapsulate points within hyperspheres (circles in 2D), enabling efficient nearest neighbor searches by leveraging the properties of these spheres.

  In a ball tree, the root node represents a circle encompassing all points in the dataset, with its center typically being the centroid and its radius extending to the farthest point from this center. The structure then undergoes recursive partitioning: each child node represents a smaller circle containing a subset of the parent node's points. This partitioning minimizes overlap between sibling nodes and ensures that each point is contained within exactly one child node. Additionally, the algorithm balances the number of points across nodes to keep the tree efficient.

  During nearest neighbor searches, the algorithm compares the distance between a new point and the center of each node's circle. If the point falls within a node's radius, the search continues within that node's children. Ball trees offer efficiency in high-dimensional data and an average-case time complexity of O(logn) for nearest neighbor searches. By encapsulating points within circles, they reduce overlap, speeding up search queries by eliminating irrelevant areas swiftly.

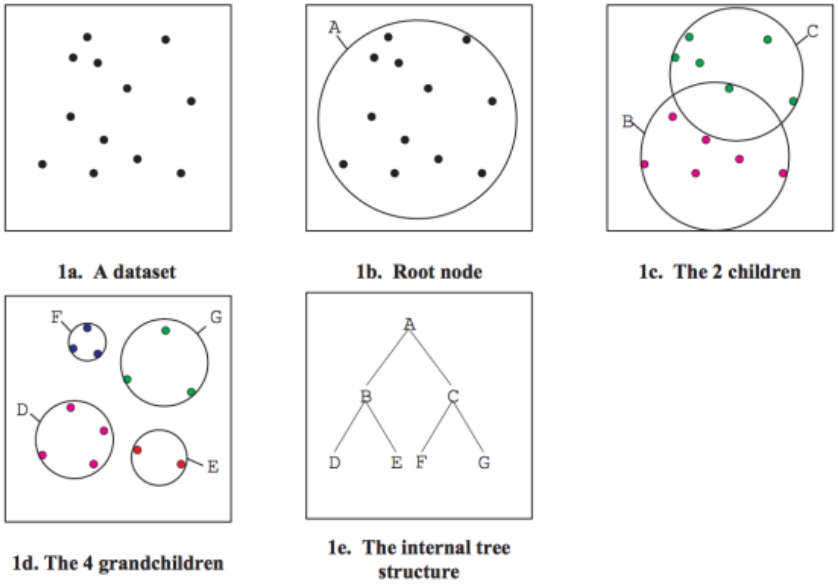



Figure 6: Example of a Ball Tree Structure for Efficient k-Nearest Neighbor Searches, as shown in [4].

Among these, the Incremental Algorithm and the Divide-and-Conquer Algorithm are the most commonly used for maintaining dynamic Voronoi diagrams. While we introduced the Divide-and-Conquer Algorithm in section 6.3, we now provide a more detailed examination of the Incremental Algorithm.

## 7.2 Incremental Algorithm

1. **Identify the Neighboring Points Whose Voronoi Cells Could Potentially Be Affected by the New Point:**
   When a new point $p_{i+1}$ is added to the Voronoi diagram, only the Voronoi cells of the neighboring points are affected. To find these neighboring points, we use efficient spatial search methods like k-d trees or ball trees. These data structures allow us to perform a nearest neighbor search in $O(\log n)$ time, where $n$ is the current number of points in the diagram.

   By finding the nearest neighbors of $p_{i+1}$, we can determine the points whose Voronoi cells need to be updated. These neighboring points are the ones closest to $p_{i+1}$, and their regions will be "cut" by the new bisectors introduced by the addition of $p_{i+1}$.

   The complexity of identifying the neighbors in the Voronoi diagram using a k-d tree or ball tree is $O(\log n)$, ensuring efficient neighbor searches even as the diagram grows.

2. **Compute the Bisector Between $p_{i+1}$ and These Neighboring Points**
   For each neighboring point $p_j$, we need to calculate the **perpendicular bisector** between the new point $p_{i+1}$ and $p_j$. This bisector is the line that divides the space equally between the two points and will form part of the updated Voronoi cell boundaries.

   Given two points $p_{i+1}(x_{i+1}, y_{i+1})$ and $p_j(x_j, y_j)$, the bisector is calculated in three steps:

   (a) **Find the Midpoint**
   The midpoint $M$ of the line segment connecting $p_{i+1}$ and $p_j$ is calculated as follows:

   $$M = \left( \frac{x_{i+1} + x_j}{2}, \frac{y_{i+1} + y_j}{2} \right)$$

   This point represents the location where the bisector will pass through, as it divides the segment into two equal halves.

   (b) **Find the Slope of the Perpendicular Bisector**
   First, calculate the slope of the line segment between $p_{i+1}$ and $p_j$:

   $$\text{slope of the line segment} = \frac{y_j - y_{i+1}}{x_j - x_{i+1}}$$

   The slope of the bisector will be the negative reciprocal of the above slope, as the bisector is perpendicular to the line segment:

   $$\text{slope of the bisector} = -\frac{x_j - x_{i+1}}{y_j - y_{i+1}}$$

   If $y_j = y_{i+1}$, the bisector will be a vertical line. Conversely, if $x_j = x_{i+1}$, the bisector will be a horizontal line.

   (c) **Equation of the Perpendicular Bisector**
   Using the midpoint $M$ and the slope of the bisector, we can write the equation of the bisector in point-slope form:

   $$y - \frac{y_{i+1} + y_j}{2} = -\frac{x_j - x_{i+1}}{y_j - y_{i+1}} \left( x - \frac{x_{i+1} + x_j}{2} \right)$$

   This equation represents the bisector $b_{i+1,j}$, which is the line that divides the space between $p_{i+1}$ and $p_j$ equally.

Since the bisector computation only involves simple algebraic operations (midpoint and slope calculations), the complexity of computing the bisector between any two points is constant, i.e., $O(1)$.

3. **Update the Voronoi Diagram by "Cutting" New Regions Using the Bisectors**
   Once the bisectors are computed, the Voronoi diagram must be updated by "cutting" the existing regions. This step involves adjusting the affected Voronoi cells to incorporate the new point $p_{i+1}$.

   For each neighboring point, the region of the Voronoi cell is divided by the new bisector, and a new region is created for the point $p_{i+1}$. The "cutting" process updates the vertices and edges of the Voronoi diagram to ensure that each point's region still contains all points closer to it than to any other point.

   In the worst case, updating the Voronoi diagram requires adjusting the Voronoi regions of all neighboring points. Since the number of neighbors is typically constant(*), this operation has a constant time complexity for each insertion.

   Thus, the complexity of updating the Voronoi diagram is $O(1)$ in the average case. However, in degenerate configurations where many points are affected, the worst-case complexity could be $O(n)$, though such cases are rare.

   (*): 'the number of neighbors is typically constant' based on [7, 8]. These references discuss how, due to the geometric layout and distribution constraints of 2D space, Voronoi vertices typically have degree 3, resulting in cells with approximately six neighboring regions on average.

### 7.3 Overall Complexity

The overall complexity of adding a point to the Voronoi diagram using this incremental algorithm can be broken down as follows:

- Identifying neighboring points: $O(\log n)$
- Computing bisectors: $O(1)$ per neighbor
- Updating the Voronoi diagram: $O(1)$ on average, $O(n)$ in the worst case

In total, the average-case complexity of adding a point to the Voronoi diagram is $O(\log n)$, while the worst-case complexity is $O(n)$, though this occurs only in special cases with degenerate configurations.

## 8 Continuing Research: Enhancing Dynamic Voronoi Diagram Efficiency

Our research began by evaluating dynamic Voronoi diagram algorithms with a focus on optimizing point update mechanisms. We demonstrated that existing algorithms can achieve efficient complexities, notably reaching O(log(n)) for incremental updates. we explored additional strategies to further improve the complexity. Initially, we examined various enhancements to the update process, such as refining neighbor searches and restructuring the update logic. However, these methods ultimately yielded minimal gains in complexity due to the intrinsic dependencies on the number of points n within the update calculations. As a result, we redirected our focus toward reducing the effective value of n without compromising the accuracy of the Voronoi diagram. our direction is leveraging Spectral Graph Theory on Voronoi-based graphs. This approach aims to examine the underlying structure of Voronoi diagrams by analyzing the eigenvalues and eigenvectors of associated adjacency and Laplacian matrices. Through spectral methods, we can gain insights into key graph properties, like clustering potential and connectivity robustness, that might allow us to group or approximate points, thus reducing n while retaining essential structural characteristics

### 8.1 Applying Spectral Analysis to Voronoi-Based Graphs

By constructing Voronoi-based graphs where each seed point represents a node and edges connect neighboring cells, we can apply spectral graph analysis to reveal insights about spatial partitioning. This involves studying the adjacency matrix A and the Laplacian matrix L of the graph:

1. **Adjacency Matrix A:** his matrix captures the connectivity between Voronoi cells based on shared boundaries. Given the sparsity of Voronoi-based graphs, A remains sparse, even with large n, supporting efficient computation.

2. **Laplacian Matrix L:** Defined as $L = D - A$, where D is the degree matrix(The degree matrix D is a diagonal matrix where each entry $D_{ii}$ represents the degree of vertex i, or the number of edges connected to that vertex, within the Voronoi-based graph), the Laplacian's eigenvalues encode information about the graph's connectivity. Notably, the second smallest eigenvalue (the Fiedler value) represents the algebraic connectivity, indicating how well-connected the graph is.

Using spectral clustering methods, we can potentially reduce the effective number of nodes by clustering highly interconnected nodes, minimizing computational overhead without drastically affecting diagram accuracy.
We can explore several potential directions: by identifying spectral clusters, we can selectively update regions of the Voronoi diagram that have the greatest impact on connectivity. Additionally, by reducing the number of active points n at regular intervals t, we could further streamline computations and improve efficiency.

# Final Remarks

We hope you enjoy our work and that it opens new perspectives on dynamic Voronoi diagrams and their applications.
"Discovery consists of seeing what everybody has seen and thinking what nobody has thought." —Albert Szent-Györgyi

# References


[1] Asano, A., Guibas, L. J., Hershberger, J. M., Lee, D. T., & Mount, D. M. (1991). *An Efficient Incremental Algorithm for Maintaining Voronoi Diagrams.* IEEE Transactions on Computers, 40(6), 744-754.

[2] Lee, D. T., & Schachter, E. (1980). *An Optimal Incremental Algorithm for Constructing Voronoi Diagrams.* Journal of Algorithms, 1(1), 1-13.

[3] Overmars, M. H., & van Leeuwen, J. H. (1981). *Maintaining Dynamic Geometric Structures.* Journal of Algorithms, 2(2), 130-169.

[4] Dong, W., Wang, Q., Josephson, W., Charikar, M., & Li, K. (2011). *Parallel k Nearest Neighbor Graph Construction Using Tree-Based Data Structures.* Proceedings of the 20th International Conference on World Wide Web.

[5] Chung, F. R. K. (1997). *Spectral Graph Theory.* American Mathematical Society.

[6] Bentley, J. L. (1975). *Multidimensional Binary Search Trees Used for Associative Searching.* Communications of the ACM, 18(9), 509-517.

[7] Aurenhammer, F. (1991). *Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure.* ACM Computing Surveys, 23(3), 345-405.

[8] Okabe, A., Boots, B., Sugihara, K., & Chiu, S. N. (2000). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons.

[9] Green, P. J., & Sibson, R. (1977). *Computing Dirichlet Tessellations in the Plane.* The Computer Journal, 21(2), 168-173.

[10] Guruprasad, K. R., & Ghose, D. (2007). *Multi-Agent Search Using Voronoi Partitions.* International Journal of Robotics Research.

[11] Houle, M. E., & Sommer, C. (2006). *Approximate Shortest Path Queries in Graphs Using Voronoi Duals.* National Institute of Informatics.

[12] Li, Y., Dong, T., Bikdash, M., & Song, Y. D. (2005). *Path Planning for Unmanned Vehicles Using Ant Colony Optimization on a Dynamic Voronoi Diagram.* Proceedings of the 2005 IEEE International Conference on Robotics and Automation.

[13] Hu, J. W., Wang, M., Zhao, C. H., Pan, Q., & Du, C. (2020). *Formation Control and Collision Avoidance for Multi-UAV Systems Based on Voronoi Partition.* Science China Technological Sciences, 63(1), 65-72.

[14] Sacristán, V. (2020). *Algorithms for Constructing Voronoi Diagrams.* Universitat Politècnica de Catalunya.

[15] Niu, H., Savvaris, A., Tsourdos, A., & Ji, Z. (2019). *Voronoi-Visibility Roadmap-Based Path Planning Algorithm for Unmanned Surface Vehicles.* The Journal of Navigation, 72(4), 850-874.