Voronoi Diagram

adzetto

June 18, 2024

Abstract

This project's goal is to use a collection of uniformly distributed points on a plane to construct the Voronoi cell that corresponds to a point at the origin. Making points, finding distances, finding the Voronoi cell, and then visualising the data are all part of this process. Our research shows how to do this in Python and identifies its strengths and weaknesses as a practical tool.

An animation package called Manim was used to produce a simulation that would further show how the Voronoi diagram was generated. You can see the process of building the Voronoi cell for a point at the origin in this simulation. The following link will take you to YouTube, where you can see the simulation: `https://youtu.be/6qEfmROZ5AA`, `https://youtu.be/hTJOVJYhSf8`, `https://youtu.be/4JLaDXvqGDo`.

Note. Code formatting is in accordance with `PEP 8 - Style Guide for Python Code`; `https://peps.python.org/pep-0008/` requirements. Guidelines for Python code organisation were drafted by Guido van Rossum, Barry Warsaw, and Alyssa Coghlan and are outlined in these guidelines. Following the guidelines laid down by *PEP 8,* the code is now structured in this format.

Contents

Chapter 1

Introduction

1.1 Purpose

The main goal of this project is to make the Voronoi cell that goes with a point at the origin in a plane that is filled with randomly placed points. The project includes several important steps: picking random points, figuring out how far these points are from the starting point, deciding where the edges of the Voronoi cell will be, and finally, showing the Voronoi cell on a graph. The study aims to give a full picture of the geometric features of Voronoi graphs and how they can be used in real life.

1.2 Motivation

There are many uses for voronoi diagrams in many fields, such as computer science, biology, and geography. They are an important part of computational geometry. The need to understand the academic and practical sides of these geometric shapes is what led to this project. In the area of computer science, Voronoi maps are used in images, location analysis, and methods for finding the shortest path. In geography, they are used to make maps and plan spaces, and in biology, they can be used to make models of cell structures and regions. By using a Voronoi model, this project aims to help people understand these uses better and lay the groundwork for more study and development.

1.3 Aim

The project's goal is to create a Python programme that can randomly place points in a plane, find the Voronoi cell that corresponds to a point at the origin, and draw that cell. To correctly calculate and see the Voronoi cell, this requires using a number of different computer methods and programmes.

1.4 Objectives

To achieve the aim of this project, the following specific objectives have been established:

- To generate a set of randomly distributed points within a two-dimensional plane.
- To compute the distances from each of these points to a fixed point at the origin.
- To determine the boundaries of the Voronoi cell associated with the origin based on the calculated distances.
- To visualize the Voronoi cell along with the randomly generated points using appropriate plotting techniques.

1.5 Scope

Python and its tools, such as numpy and matplotlib, will be used in this project to make Voronoi diagrams and show them. Using this method, random points are made in polar coordinates, then they are changed to Cartesian coordinates and the Voronoi cell for the origin is calculated. For a better understanding of the Voronoi cell's structure, the results of this application will be shown in image form. This project is mostly about making a simple solution for a fixed point at the origin, but the methods and results can be used in more complicated situations with bigger datasets and different point distributions, which means the study can be used in more areas.

1.6 Significance

For many useful situations, it is important to know how Voronoi models are put together and what their features are. This project not only helps students learn how to understand geometric shapes, but it also lays the groundwork for more advanced uses in science and business. The study's findings can be used to make the best use of space resources, make computer programmes run faster, and create better models of nature events. These findings will greatly assist progress in both the theory and applied sciences.

1.7 Structure of the Report

The report is organized as follows:

- **Chapter 2: Literature Review** - Summarises what has already been written about Voronoi diagrams, including their theory bases and real-world uses. It has parts about the past of Voronoi diagrams and other works in the same field.

- **Chapter 3: Proposed System Analysis & Design** - Gives more information about the problem description, analysis, viability study, requirement analysis, and system design. Included in this chapter is a diagram that shows the steps that were taken to create and display the Voronoi cell.
- **Chapter 4: Technology Implementation & Testing** - It talks about the technologies and tools that were used in the project, like Random, Numpy, Matplotlib, and Python. It includes the Python code and gives a thorough explanation of how to make a Voronoi map.
- **Chapter 5: Application, Advantages & Limitations** - Talks about the different ways Voronoi diagrams can be used, the benefits of using them in different areas, and the problems with the way they are currently implemented.
- **Chapter 6: Conclusion and Future Work** - Summarises the project's results, talks about what those results mean, and offers possible areas for more study and changes in the future.

Chapter 2

Literature Review

2.1 Background History

An essential building block of computational geometry, voronoi diagrams divide a plane into separate areas according to their distance from a predetermined set of points called generators. So, let $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$ be a set of n distinct points. The Voronoi cell $V(p_i)$ associated with the generator $p_i \in P$ is defined as:

$$V(p_i) = \{x \in \mathbb{R}^2 \mid \forall j \neq i, \|x - p_i\| \leq \|x - p_j\|\}$$

where $\|\cdot\|$ denotes the Euclidean norm. These regions, or cells, have broad applicability across multiple disciplines.

2.2 Related Works

Theoretical developments and practical uses of voronoi diagrams are covered extensively in the literature. Some notable innovations include iterative methods for building centroidal Voronoi tessellations (CVTs) and algorithms for efficient calculation of Voronoi diagrams (e.g., Fortune's sweepline algorithm). Let $\Omega \subseteq \mathbb{R}^2$ be a domain, and let $\rho : \Omega \to \mathbb{R}$ be a density function. The objective in constructing a CVT is to find a set of points $P \subset \Omega$ such that each point $p_i \in P$ is the centroid of its respective Voronoi cell $V(p_i)$, weighted by ρ. Formally, p_i must satisfy:

$$p_i = \frac{\int_{V(p_i)} x\rho(x)\, dx}{\int_{V(p_i)} \rho(x)\, dx}$$

Geometric models are useful in many fields; for example, spatial analysis may help with location optimisation issues, and the biological sciences can use them to better understand cellular structures and patterns.

2.3 Summary & Discussion

Voronoi diagrams and their centroidal variations are fundamentally important for studying their computational characteristics and applications. This knowledge makes it easier to create effective algorithms and put them into practice in a variety of real-world contexts.

Chapter 3

Proposed System Analysis & Design

3.1 Problem Statement

Given a point located at the origin $O \in \mathbb{R}^2$ and a set of randomly distributed points $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$, the task is to determine the Voronoi cell associated with the origin. Formally, we seek to compute the region:

$$V(O) = \{x \in \mathbb{R}^2 \mid \forall p_i \in P, \|x - O\| \leq \|x - p_i\|\}$$

This involves generating points, calculating distances, and employing algebraic methods to delineate $V(O)$.

3.2 Analysis

Python is used to create the points and calculate the required distances. The process of analysis is as follows: first, in polar coordinates, create P. Then, transform it to Cartesian coordinates. Finally, calculate the distances in Euclidean terms from each point in P to the origin. How is the distance function defined:

$$d_i = \|p_i - O\| = \sqrt{x_i^2 + y_i^2}$$

for each $p_i = (x_i, y_i) \in P$. The Voronoi cell $V(O)$ is constructed by finding the perpendicular bisectors of the line segments $[O, p_i]$ and determining their intersections.

3.3 Feasibility Study

Technical Feasibility: No specialised hardware or software is required since the project uses ordinary Python libraries like as `numpy`, `matplotlib`, and `random`.

Economic Feasibility: The cost is minimal, requiring only a computer with Python installed.

Behavioral Feasibility: The project is straightforward and accessible, making it suitable for educational and research purposes.

3.4 Requirement Analysis

The necessary tools include:

- `Python`: For implementation.
- `numpy`: For numerical computations.
- `matplotlib`: For plotting results.
- `random`: For generating random points.

3.5 System Design

The design involves the following steps:

1 | Generate random points P in polar coordinates and convert to Cartesian coordinates.

2 | Compute distances $d_i = \|p_i - O\|$ for each $p_i \in P$.

3 | Identify the closest points to the origin and calculate the perpendicular bisectors of the segments $[O, p_i]$.

4 | Determine the intersection points of these bisectors to form the boundaries of $V(O)$.

5 | Plot the points and the Voronoi cell using `matplotlib`.

Mathematically, the perpendicular bisector of $[O, p_i]$ can be described as:

$$\text{Bisector}(O, p_i) : (x - \frac{x_i}{2})x_i + (y - \frac{y_i}{2})y_i = 0$$

Solving the system of linear equations given by the bisectors will yield the vertices of the Voronoi cell.

3.5.1 Flowchart

For now, I will just leave this here with a simple flowchar showing the process. I will give more details in the Implementation part.

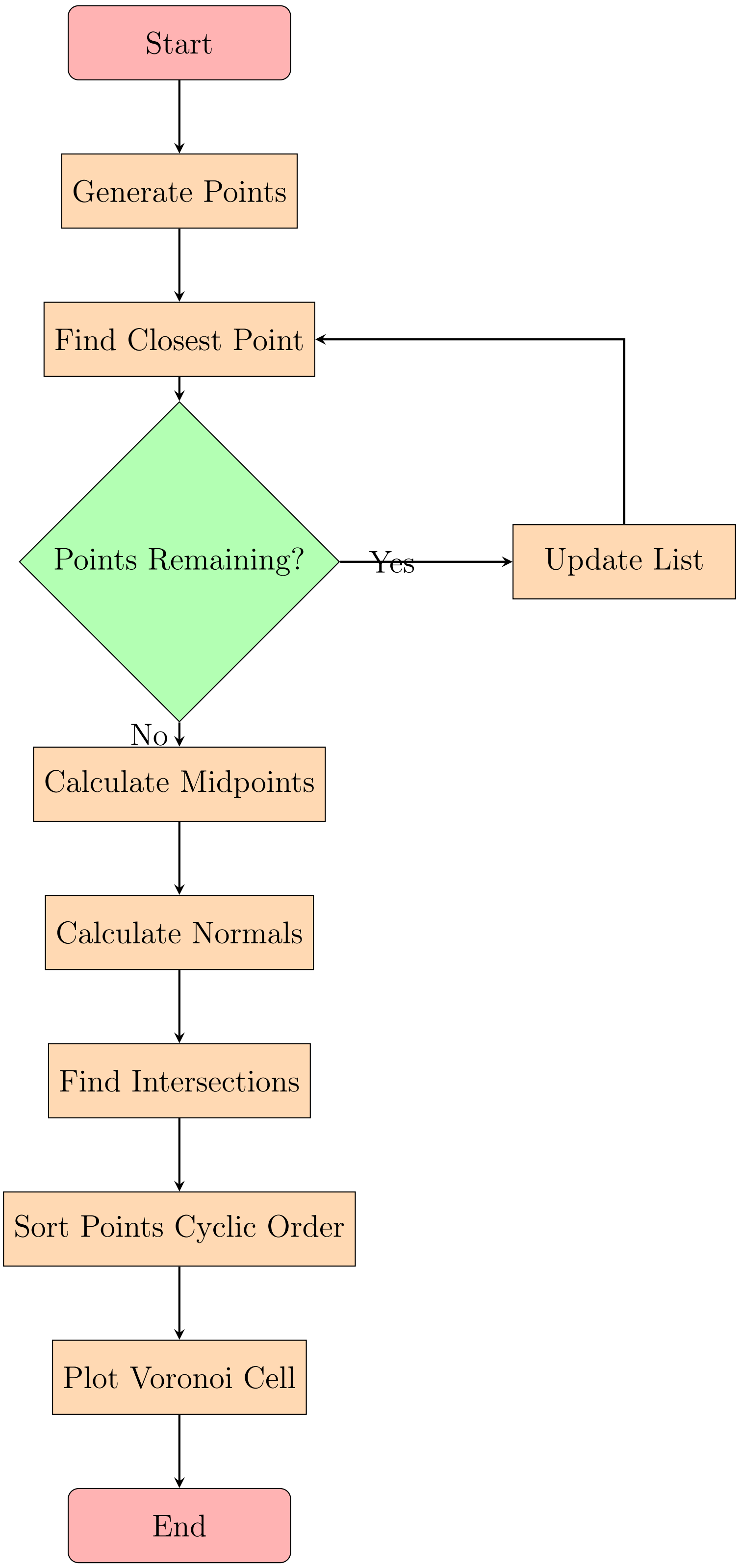


Figure 3.1: Flowchart of the Voronoi Diagram Generation Process

Chapter 4

Technology Implementation & Testing

4.1 Technology

- **Python**: A high-level programming language used for implementing the project.
- **Numpy**: A library for numerical operations in Python.
- **Matplotlib**: A library for plotting graphs in Python.
- **Random**: A module for generating random numbers in Python.

4.2 Implementation

4.2.1 Point Generation Function

Purpose

The `generate_points` function generates a specified number of points within given ranges for the radial and angular coordinates. These points are evenly distributed around the center and are transformed from polar to Cartesian coordinates.

Arguments & Returns

Amount of points to create, denoted as n. r_{min} is the smallest radius. the greatest radius, denoted as r_{max}. θ_{min}: The angle with the smallest value (of degrees). In degrees, θ_{max} represents the greatest angle. This function takes an array of tuples as input and returns a new array with the distance from the origin as well as the x and y coordinates of each set.

Algorithm

Algorithm 1 Generate Random Points in Polar Coordinates

function GENERATE_POINTS($n, r_{min}, r_{max}, \theta_{min}, \theta_{max}$)
    $\mathcal{P} \leftarrow \varnothing$
    for $i \leftarrow 1$ to n **do**
        $r \leftarrow \text{random.uniform}(r_{min}, r_{max})$
        $\theta \leftarrow \text{random.uniform}(\theta_{min}, \theta_{max})$
        $x \leftarrow r \cdot \cos(\theta)$
        $y \leftarrow r \cdot \sin(\theta)$
        $d \leftarrow \sqrt{x^2 + y^2}$
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{(x, y, d)\}$
    end for
    return $\mathcal{P}$
end function

Explanation

The function converts polar coordinates (r, θ) to Cartesian coordinates (x, y) using:

$$x = r \cdot \cos(\theta)$$

$$y = r \cdot \sin(\theta)$$

The distance d from the origin is calculated as:

$$d = \sqrt{x^2 + y^2}$$

Python Code

points.py

```
import random
import math

def generate_points(n, r_min, r_max, theta_min, theta_max):
    points = []
    for _ in range(n):
        r = random.uniform(r_min, r_max)
        theta = random.uniform(theta_min, theta_max)
        x = r * math.cos(math.radians(theta))
        y = r * math.sin(math.radians(theta))
        distance = math.sqrt(x**2 + y**2)
        points.append((x, y, distance))
    return points
```

Listing 4.1: Function to Generate Points

4.2.2 Generating Points in Different Angular Segments

Purpose

To generate points inside defined angular segments in such a way that they are evenly distributed around the origin.

Python Code

main.py

```
points_first = (
    generate_points(n=5, r_min=2.5, r_max=15, theta_min=5,
    theta_max=85) +
    generate_points(n=5, r_min=2.5, r_max=15, theta_min=95,
    theta_max=175) +
    generate_points(n=5, r_min=2.5, r_max=15, theta_min=185,
    theta_max=265) +
    generate_points(n=5, r_min=2.5, r_max=15, theta_min=275,
    theta_max=355)
)
```

Listing 4.2: Generating Points in Different Angular Segments

4.2.3 Finding the Closest Point Function

Purpose

By removing previously found closest points, the `find_closest_point` function finds the point in a list of points that is closest to a reference point P_0.

Arguments & Returns

The set of points is denoted by $\mathcal{P}$. Point P_0 is the starting point. In earlier rounds, the locations that were determined to be closest were added to the $\mathcal{C}$ list. Finds the point nearest to P_0 and returns its index and coordinates.

Algorithm

Algorithm 2 Find the Closest Point

function FIND_CLOSEST_POINT$(\mathcal{P}, P_0, \mathcal{C})$
    $\mathcal{P}_{\text{array}} \leftarrow \text{array}([\mathbf{p}[:2] \text{ for } \mathbf{p} \in \mathcal{P}])$
    $P_{0,\text{array}} \leftarrow \text{array}(P_0)$
    $\mathcal{D} \leftarrow \text{array}([\|\mathbf{p} - P_{0,\text{array}}\| \text{ for } \mathbf{p} \in \mathcal{P}_{\text{array}}])$
    for $\mathbf{c} \in \mathcal{C}$ **do**
        if $\mathbf{c} \in \mathcal{P}$ **then**
            $index \leftarrow \text{index}(\mathbf{c})$
            $\mathcal{D}[index] \leftarrow \infty$
        end if
    end for
    $i_{\text{min}} \leftarrow \text{argmin}(\mathcal{D})$
    return $(i_{\text{min}}, \mathcal{P}[i_{\text{min}}])$
end function

Explanation

Consider the set of points $\mathbf{p}_i = (x_i, y_i, d_i)$ where each point $\mathcal{P} \subset \mathbb{R}^2 \times \mathbb{R}$ contains the coordinates and distance from the origin. In prior rounds, the set of points that were determined to be closest was denoted as $\mathcal{C} \subset \mathcal{P}$ and the reference point was denoted as $P_0 = (x_0, y_0) \in \mathbb{R}^2$. Finding the point in $\mathcal{P} \setminus \mathcal{C}$ that is closest to P_0 is the goal. The collection of coordinates of points in $\mathcal{P}$ is defined as the array $\mathcal{P}_{\text{array}}$.

$$\mathcal{P}_{\text{array}} = \{\mathbf{p}[:2] \mid \mathbf{p} \in \mathcal{P}\}$$

Let $P_{0,\text{array}}$ be the array representation of the reference point P_0:

$$P_{0,\text{array}} = \text{array}(P_0)$$

Compute the Euclidean distances $\mathcal{D}$ from P_0 to each point in $\mathcal{P}_{\text{array}}$:

$$\mathcal{D} = \{\|\mathbf{p} - P_{0,\text{array}}\| \mid \mathbf{p} \in \mathcal{P}_{\text{array}}\}$$

For each point $\mathbf{c} \in \mathcal{C}$, if $\mathbf{c} \in \mathcal{P}$, set the corresponding distance in $\mathcal{D}$ to ∞ to exclude it from consideration:

$$\text{For each } \mathbf{c} \in \mathcal{C}, \text{ if } \mathbf{c} \in \mathcal{P}, \text{ set } \mathcal{D}[\text{index}(\mathbf{c})] = \infty$$

Determine the index i_{min} of the minimum distance in $\mathcal{D}$:

$$i_{\text{min}} = \text{argmin}(\mathcal{D})$$

Return the index and coordinates of the closest point:

$$\text{Return } (i_{\text{min}}, \mathcal{P}[i_{\text{min}}])$$

After selecting all points and setting their distances to infinity, the function finds the one with the smallest Euclidean distance to P_0 and returns it.

Python Code

find_closest.py

```python
import numpy as np
from numpy.linalg import norm

def find_closest_point(points, P_0, closest_points):
    points_array = np.array([point[:2] for point in points])
    P_0_array = np.array(P_0)

    distances = np.array([norm(point - P_0_array) for point
  in points_array])
    for point in closest_points:
        if point in points:
            index = points.index(point)
            distances[index] = float('inf')

    closest_point_index = np.argmin(distances)
    return closest_point_index, points[closest_point_index]
```

Listing 4.3: Function to Find Closest Point

4.2.4 Unit Vector Calculation Function

Purpose

Geometric algorithms that need direction and normalisation rely on the `find_unit_vector` function, which computes the unit vector from one point to another.

Arguments & Return

P_{from}: The starting point. P_{to}: The ending point. Returns the unit vector from P_{from} to P_{to}.

Algorithm

Algorithm 3 Find Unit Vector

Input: Points $P_{\text{from}} = (x_{\text{from}}, y_{\text{from}})$ and $P_{\text{to}} = (x_{\text{to}}, y_{\text{to}})$
Output: Unit vector $\hat{\mathbf{v}}$
$\mathbf{v} \leftarrow \begin{pmatrix} x_{\text{to}} - x_{\text{from}} \\ y_{\text{to}} - y_{\text{from}} \end{pmatrix}$ ▷ Compute the difference vector
$\|\mathbf{v}\| \leftarrow \sqrt{(x_{\text{to}} - x_{\text{from}})^2 + (y_{\text{to}} - y_{\text{from}})^2}$ ▷ Calculate the Euclidean norm of $\mathbf{v}$
if $\|\mathbf{v}\| = 0$ **then**
    return $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ▷ Return zero vector if norm is zero
else
    return $\mathbf{v}/\|\mathbf{v}\|$ ▷ Return the normalized vector
end if

Mathematical Explanation

Given two points $P_{\text{from}} = (x_{\text{from}}, y_{\text{from}})$ and $P_{\text{to}} = (x_{\text{to}}, y_{\text{to}})$:

$$\mathbf{v} = \begin{pmatrix} x_{\text{to}} - x_{\text{from}} \\ y_{\text{to}} - y_{\text{from}} \end{pmatrix}$$

$$\|\mathbf{v}\| = \sqrt{(x_{\text{to}} - x_{\text{from}})^2 + (y_{\text{to}} - y_{\text{from}})^2}$$

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Python Code

vector_utils.py

```python
def find_unit_vector(P_from, P_to):
    vector = np.array(P_to[:2]) - np.array(P_from[:2])
    vector_norm = norm(vector)
    if vector_norm == 0:
        return np.array([0, 0])
    return vector / vector_norm
```

Listing 4.4: Finding the Unit Vector Between Two Points

4.2.5 Filtering Points by Dot Product

Purpose

The function `filter_points_by_dot_product` is used to sort a list of points so that only those points may be used to create a dot product that is not negative, with relation to a reference vector and a base point. Finding points that are roughly in the same direction as the reference vector from the base point is made easier using this method.

Arguments & Returns

The points to filter are referenced by the list `points`. Distance from the origin, along with x and y, is included in each point's tuple. The starting point for measuring directions is called the base point. * The reference vector is used to compute the dot product. Provides a set of points as an array whose dot product with the reference vector, when measured from the base point, is positive.

Algorithm

Algorithm 4 Filter Points by Dot Product

Input: Set of points $\mathcal{P} = \{\mathbf{p}_i = (x_i, y_i, d_i) \mid i \in \{1, \ldots, n\}\}$, base point $P_{\text{base}} = (x_{\text{base}}, y_{\text{base}}, d_{\text{base}})$, reference vector $\mathbf{v}_{\text{ref}} \in \mathbb{R}^2$
Output: Filtered set of points $\mathcal{R}$
$\mathcal{R} \leftarrow \varnothing$ ▷ Initialize the set of remaining points
for each $\mathbf{p}_i \in \mathcal{P}$ **do**
    if $\mathbf{p}_i[:2] = (x_i, y_i) = (x_{\text{base}}, y_{\text{base}})$ **then**
        continue ▷ Skip the base point itself
    end if
    $\mathbf{u}_i \leftarrow (x_i - x_{\text{base}}, y_i - y_{\text{base}})$ ▷ Compute vector from base point to current point
    $\|\mathbf{u}_i\| \leftarrow \sqrt{(x_i - x_{\text{base}})^2 + (y_i - y_{\text{base}})^2}$ ▷ Calculate the Euclidean norm of $\mathbf{u}_i$
    if $\|\mathbf{u}_i\| = 0$ **then**
        $\hat{\mathbf{u}}_i \leftarrow (0, 0)$
    else
        $\hat{\mathbf{u}}_i \leftarrow \mathbf{u}_i / \|\mathbf{u}_i\|$ ▷ Normalize $\mathbf{u}_i$ to get the unit vector
    end if
    dot_product $\leftarrow \mathbf{v}_{\text{ref}} \cdot \hat{\mathbf{u}}_i$ ▷ Compute the dot product of $\mathbf{v}_{\text{ref}}$ and $\hat{\mathbf{u}}_i$
    if dot_product ≥ 0 **then**
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{\mathbf{p}_i\}$ ▷ Include $\mathbf{p}_i$ in $\mathcal{R}$ if the dot product is non-negative
    end if
end for
return $\mathcal{R}$

Explanation

Bypassing the starting point, the function iteratively processes all elements in the list. After finding the unit vector that extends from the origin to the present location, it takes the dot product of this vector and the reference vector and keeps the result if it's not negative.

Python Code

filter_points.py

```python
import numpy as np
from numpy.linalg import norm

def norm(vector):
    return np.sqrt(np.sum(np.square(vector)))

def find_unit_vector(P_from, P_to):
    vector = np.array(P_to[:2]) - np.array(P_from[:2])
    vector_norm = norm(vector)
    if vector_norm == 0:
        return np.array([0, 0])
    return vector / vector_norm

def filter_points_by_dot_product(points, base_point,
    reference_vector):
    remaining_points = []
    for point in points:
        if np.array_equal(point[:2], base_point[:2]):
            continue
        unit_vector = find_unit_vector(base_point, point[:2])
        dot_product = np.dot(reference_vector, unit_vector)
        if dot_product >= 0:
            remaining_points.append(point)
    return np.array(remaining_points)
```

Listing 4.5: Filtering Points by Dot Product

4.2.6 Point Selection and Filtering Loop

Purpose

In this code snippet, the closest point to P_0 is chosen from a list of points in an iterative fashion. The remaining points are then filtered according to their dot product with a reference vector, and the process continues until they are all removed.

Algorithm

Algorithm 5 Point Selection and Filtering Loop

Input: Set of points $\mathcal{P}$, reference point P_0, and an initially empty list $\mathcal{C}$ for closest points.
while $\mathcal{P} \neq \varnothing$ **do**
  $(i_{\min}, \mathbf{p}_{\min}) \leftarrow \text{find\_closest\_point}(\mathcal{P}, P_0, \mathcal{C})$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{p}_{\min}\}$
  $\mathbf{v}_{\text{ref}} \leftarrow \text{find\_unit\_vector}(\mathbf{p}_{\min}, P_0)$
  $\mathcal{P} \leftarrow \text{filter\_points\_by\_dot\_product}(\mathcal{P}, \mathbf{p}_{\min}, \mathbf{v}_{\text{ref}})$
  Print "Selected closest point: $\mathbf{p}_{\min}$"
  if $\mathcal{P} = \varnothing$ **then**
    break
  end if
end while

Explanation

By selecting the nearest points repeatedly, the algorithm filters the points until there are no more points.

Python Code

main.py

```python
    while len(points) > 0:
        closest_point_index, closest_point = find_closest_point(
            points, P_0, closest_points
        )
        closest_points.append(closest_point)

        reference_vector = find_unit_vector(P_from=closest_point, P_to=P_0)
        points = filter_points_by_dot_product(points, closest_point, reference_vector)

        print(f"Selected closest point: {closest_point}")

        if points.size == 0:
            break
```

Listing 4.6: Point Selection and Filtering Loop

4.2.7 Midpoints and Normals Calculation

Purpose

Each point in the list of nearest points relative to P_0 has its midpoint and normal calculated using this code snippet. Next, the angular coordinates are used to order the midpoints.

Algorithm

Algorithm 6 Calculate Midpoints and Normals

Input: List of closest points $\mathcal{C}$, reference point P_0
Output: Sorted midpoints and their corresponding normals
$\mathcal{M} \leftarrow \left\{ \frac{\mathbf{p}[:2]+P_0}{2} \mid \mathbf{p} \in \mathcal{C} \right\}$ ▷ Calculate midpoints between each closest point and P_0
$\mathcal{N} \leftarrow \varnothing$ ▷ Initialize list for normals
for each $\mathbf{p} \in \mathcal{C}$ **do**
  $\mathbf{v} \leftarrow \mathbf{p}[:2] - P_0$ ▷ Compute vector from P_0 to $\mathbf{p}$
  $\mathbf{n} \leftarrow \begin{pmatrix} -v_y \\ v_x \end{pmatrix}$ ▷ Calculate the normal vector by rotating $\mathbf{v}$ by 90 degrees
  $\hat{\mathbf{n}} \leftarrow \frac{\mathbf{n}}{\|\mathbf{n}\|}$ ▷ Normalize the normal vector
  $\mathcal{N} \leftarrow \mathcal{N} \cup \{\hat{\mathbf{n}}\}$ ▷ Append the unit normal to the list
end for
$\mathcal{M} \leftarrow \mathrm{array}(\mathcal{M})$
$\theta \leftarrow \arctan 2(\mathcal{M}[:,1], \mathcal{M}[:,0])$ ▷ Calculate angles of midpoints for sorting
indices $\leftarrow$ custom_argsort(θ)
$\mathcal{M} \leftarrow \mathcal{M}[\mathrm{indices}]$
$\mathcal{N} \leftarrow \mathrm{array}(\mathcal{N})[\mathrm{indices}]$ ▷ Sort midpoints and normals based on calculated angles
return $(\mathcal{M}, \mathcal{N})$

Explanation

With the reference point being $P\_0 \in \mathbb{R}^2$, let $\mathcal{C} \subset \mathbb{R}^2 \times \mathbb{R}$ represent the set of nearest points. First, we need to find the midpoints ($\mathcal{M} = \left\{ \frac{\mathbf{p}[:2]+P\_0}{2} \mid \mathbf{p} \in \mathcal{C} \right\}$ between each point in $\mathcal{C}$ and P_0, which is represented by $\mathcal{M}$. Then, we must calculate the appropriate normal vectors. Assuming that $\mathbf{p}[:2] - P\_0$ is a vector from P_0 to $\mathbf{p}$, determine the vector $\mathbf{v}$ for each $\mathbf{p} \in \mathcal{C}$. To acquire the normal vector $\mathbf{n}$ which is equal to

$$\begin{pmatrix} -v\_y \\ v\_x \end{pmatrix}$$

rotate $\mathbf{v}$ by 90 degrees. Then, normalise $\mathbf{n}$ to get the unit normal $\hat{\mathbf{n}}$. After that, arrange the normals and midpoints in descending order by sorting the indices based on the angles calculated using the arctangent function, $\theta = \arctan 2(\mathcal{M}[:,1], \mathcal{M}[:,0])$. At last, provide back the normals and sorted midpoints.

Python Code

geometry.py

```
    midpoints = [(np.array(P[:2]) + np.array(P_0)) / 2 for P in closest_points]
    normals = []
    for P in closest_points:
        vector = np.array(P[:2]) - np.array(P_0)
        normal = np.array([-vector[1], vector[0]])
        unit_normal = normal / norm(normal)
        normals.append(unit_normal)

    midpoints = np.array(midpoints)
    angles = np.arctan2(midpoints[:, 1], midpoints[:, 0])
    sorted_indices = custom_argsort(angles)
    midpoints = midpoints[sorted_indices]
    normals = np.array(normals)[sorted_indices]
```

Listing 4.7: Midpoints and Normals Calculation

4.2.8 Intersection Calculation with Custom Linear Solver

Purpose

Using a proprietary linear solver for the system of equations, this function determines the intersection point of two lines, where each line is characterised by a midpoint and a normal vector.

Algorithm

Algorithm 7 Find Intersection of Two Lines with Custom Linear Solver

Input: Midpoints $\mathbf{m}_1, \mathbf{m}_2$ and normals $\mathbf{n}_1, \mathbf{n}_2$ of two lines
Output: Intersection point $\mathbf{I}$, or **None** if lines are parallel
$A \leftarrow \begin{pmatrix} \mathbf{n}_1 & -\mathbf{n}_2 \end{pmatrix}^\top$
$b \leftarrow \mathbf{m}_2 - \mathbf{m}_1$ ▷ Construct the linear system $A\mathbf{x} = b$
if $\det(A) = 0$ **then**
    return None ▷ The lines are parallel, no intersection
else
    $\mathbf{x} \leftarrow \text{linalg\_solve}(A, b)$ ▷ Solve for $\mathbf{x}$ using the custom solver
    $\mathbf{I} \leftarrow \mathbf{m}_1 + x_1\mathbf{n}_1$ ▷ Calculate intersection point
    return I
end if

Custom Linear Solver Algorithm

Algorithm 8 Custom Linear Solver (Gaussian Elimination)

Input: Matrix $A \in \mathbb{R}^{n \times n}$, Vector $\mathbf{b} \in \mathbb{R}^n$
Output: Solution vector $\mathbf{x} \in \mathbb{R}^n$
Let $M \leftarrow [A \mid \mathbf{b}]$ ▷ Augmented matrix
Let $n \leftarrow$ number of rows of A
for $k \leftarrow 1$ to n **do** ▷ Forward elimination
  Find $p \leftarrow \arg\max_{i=k,\ldots,n} |M_{ik}|$
  if $p \neq k$ **then**
    Swap rows k and p in M
  end if
  for $i \leftarrow k+1$ to n **do**
    Let factor $\leftarrow \frac{M_{ik}}{M_{kk}}$
    for $j \leftarrow k$ to $n+1$ **do**
      $M_{ij} \leftarrow M_{ij} - \text{factor} \cdot M_{kj}$
    end for
  end for
end for
Let $\mathbf{x} \leftarrow \mathbf{0} \in \mathbb{R}^n$ ▷ Initialize solution vector
for $i \leftarrow n$ to 1 by -1 **do** ▷ Back substitution
  $x_i \leftarrow \frac{M_{i,n+1}}{M_{ii}}$
  for $j \leftarrow i-1$ to 1 by -1 **do**
    $M_{j,n+1} \leftarrow M_{j,n+1} - M_{ji} \cdot x_i$
  end for
end for
return x

Explanation

The intersection point of two lines that are described by their midpoints may be found using the `find_intersection` function. both normals $\mathbf{n}\_1, \mathbf{n}\_2$ and $\mathbf{m}\_1, \mathbf{m}\_2$. The programme uses a bespoke linear solver called `linalg_solve` to solve a system of linear equations $A\mathbf{x} = b$. In order to resolve the linear system, the `linalg_solve` function employs Gaussian elimination in conjunction with partial pivoting.

Python Code

Figure 4.1: *geometry.py*

```
import numpy as np

def find_intersection(midpoint1, normal1, midpoint2, normal2)
    :
    A = np.array([normal1, -normal2]).T
    b = np.array(midpoint2) - np.array(midpoint1)
    if np.linalg.det(A) == 0:
        return None
    intersection = linalg_solve(A, b)
    return midpoint1 + intersection[0] * normal1

def linalg_solve(A, b):
    n = len(A)
    M = [list(row) for row in A]

    for i in range(n):
        M[i].append(b[i])

    for k in range(n):
        max_row = max(range(k, n), key=lambda i: abs(M[i][k])
    )
        M[k], M[max_row] = M[max_row], M[k]
        for i in range(k + 1, n):
            factor = M[i][k] / M[k][k]
            for j in range(k, n + 1):
                M[i][j] -= factor * M[k][j]

    x = [0] * n
    for i in range(n - 1, -1, -1):
        x[i] = M[i][n] / M[i][i]
        for j in range(i - 1, -1, -1):
            M[j][n] -= M[j][i] * x[i]

    return x
```

Listing 4.8: Intersection Calculation with Custom Linear Solver

4.3 Calculation and Visualization of Intersection Points

4.3.1 Purpose

Identify important geometric elements, including the junction points of line segments described by their midpoints and normal vectors, and to display these locations graphically.

4.3.2 Algorithm for Computing Intersection Points

With respect to the midpoints, let $\mathcal{M} = \{\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_n\}$ and the normal vectors, $\mathcal{N} = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_n\}$. The aim is to find the sites where these midpoints and normals join to produce successive line segments.

Define the set of intersection points $\mathcal{I}$ as follows:

$$\mathcal{I} = \{\mathbf{I}_i \mid \mathbf{I}_i = \text{find\_intersection}(\mathbf{m}_i, \mathbf{n}_i, \mathbf{m}_{i+1}, \mathbf{n}_{i+1}),\ i = 1, \ldots, n\}$$

where $\mathbf{m}_{n+1} \equiv \mathbf{m}_1$ and $\mathbf{n}_{n+1} \equiv \mathbf{n}_1$.

4.3.3 Python Code for Intersection Points Calculation and Visualization

Figure 4.2: *visualization.py*

```python
import numpy as np
import matplotlib.pyplot as plt

intersection_points = []
for i in range(len(midpoints)):
    next_index = (i + 1) % len(midpoints)
    intersection = find_intersection(
        midpoints[i], normals[i], midpoints[next_index],
    normals[next_index]
    )
    if intersection is not None:
        intersection_points.append(intersection)
```

Listing 4.9: Intersection Points Calculation and Visualization

Figure 4.3: *visualization.py*

```
plt.figure(figsize=(10, 8))
plt.scatter([point[0] for point in points_first],[point[1]
    for point in points_first],color="gray",label="First
    Generated")
all_points = closest_points + [P_0]
x_coords = [point[0] for point in all_points]
y_coords = [point[1] for point in all_points]
plt.scatter(x_coords, y_coords, color="green", label="
    Selected Points")

for point in closest_points:
    plt.plot([P_0[0], point[0]], [P_0[1], point[1]], "gray",
    linestyle="dotted")
plt.scatter(
    [point[0] for point in midpoints],
    [point[1] for point in midpoints],
    color="blue",
    label="Midpoints",
)
```

Listing 4.10: Intersection Points Calculation and Visualization

Figure 4.4: *visualization.py*

```
intersection_points = np.array(intersection_points)
if intersection_points.size > 0:
    plt.scatter(intersection_points[:, 0],intersection_points
    [:, 1],color="purple",label="Intersections",)

    for i in range(len(intersection_points)):
        next_index = (i + 1) % len(intersection_points)
        plt.plot([intersection_points[i][0],
    intersection_points[next_index][0]],[intersection_points[i
    ][1], intersection_points[next_index][1]], color="purple")

plt.xlabel("X Coordinates")
plt.ylabel("Y Coordinates")
plt.title("Voronoi Cell and Points")
plt.grid(True)
plt.legend()
plt.show()
```

Listing 4.11: Intersection Points Calculation and Visualization

Explanation

The code determines the intersection locations of line segments defined by sequential midpoints and normals given a collection of midpoints (mathcalM) and normal vectors (mathcalN). By using the `find_intersection` function, every site of intersection is identified. We use Matplotlib to display the points, which include the original set as well as the nearest points, midpoints, and intersecting points.

Chapter 5

Application, Advantages & Limitations

5.1 Applications

The process of making a Voronoi diagram can be used for many things, such as spatial analysis and geographic mapping, transportation and logistics optimisation problems, and modelling of ecosystems and areas in the natural sciences, like biology and ecology.

5.2 Advantages

A Voronoi diagram is useful in many fields, such as geography, biology, and computer science, and has an easy-to-use method for making them. It shows clearly how points are related to each other in terms of how close they are to each other.

5.3 Limitations

However, there are some problems with the method. For example, when there are a lot of points, it takes a lot of computing power, and the accuracy of the Voronoi cell depends on how well the calculations are done.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

It is successfully shown in the project how to use randomly placed points to make a Voronoi cell connected to the origin. The application makes the Voronoi map easy to see by showing the connections between points that are close to each other.

6.2 Future Work

In the future, work could be done to make the method work better with bigger datasets, to speed up processes by using parallel processing, and to use the Voronoi diagram generation on real-world datasets for useful purposes.

Chapter 7

References

- F. Aurenhammer, H. Edelsbrunner, "An optimal algorithm for constructing the weighted Voronoi diagram in the plane," Pattern Recognition, Vol. 17, no. 2, pp. 251–257, 1984. DOI: 10.1016/0031-3203(84)90064-5.
- F. Aurenhammer, R. Klein, D.-T. Lee, "Voronoi diagrams and Delaunay triangulations," World Scientific Publishing, Singapore, 2013.
- M. Candeloro, A. M. Lekkas, A. J. Sorensen, "A Voronoi-diagram-based dynamic path-planning system for underactuated marine vessels," Control Engineering Practice, Vol. 61, pp. 41–54, 2017. DOI: 10.1016/j.conengprac.2017.01.007.
- G. Cosquer, J. F. Hangouët, "Delimitation of land and sea boundaries: Geodetic and geometric bases," FIG Working Week 2003, Paris, France, April 13–17, 2003.
- P. Dong, "Generating and updating multiplicatively weighted Voronoi diagrams for point, line and polygon features in GIS," Computers & Geosciences, Vol. 34, no. 4, pp. 411–421, 2008.
- J. Gaździcki, "Systemy informacji przestrzennej," Państwowe Przedsiębiorstwo Wydawnictw Kartograficznych, Warszawa, 1990.
- M. Gavrilova (Ed.), "A Geometry-based approach to computational intelligence," Studies in Computational Intelligence, Vol. 158, Springer, 2008.
- M. Kratko, "Heorhij Woronyj – matematyk ukraiński i polski," http://voronuy.at.ua/publ/heorhij_woronyj_matematyk_ukrainski_i_polski/1-1-0-48, 2011.
- H. Ledoux, "Computing the 3D Voronoi diagram robustly: An easy explanation," Voronoi diagrams in science and engineering, ISVD '07, 4th International Symposium, pp. 117–129, 2007.

- S. Lovacka, "The use of the Voronoi tessellation for purposes of service distribution districts delimitation (The example of the Prešov nodal region)," Folia Geographica, Vol. 12, pp. 163–171, 2008.
- A. Magnuszewski, "GIS w geografii fizycznej," Wydawnictwo Naukowe PWN, Warszawa, 1999.
- Mathworks Documentation, "Voronoi diagrams," https://www.mathworks.com/help/matlab/math/voronoi--diagrams.html.
- T. van der Putte, "Using the discrete 3D Voronoi diagram for the modelling of 3D continuous information in geosciences," Master Thesis, Utrecht University, 2009. http://www.gdmc.nl/publications/2009/3D_Voronoi_diagram.pdf.
- A. Okabe, B. Boots, K. Sugihara, S. N. Chiu, "Spatial tessellations: Concepts and applications of Voronoi diagrams," John Wiley & Sons, Chichester – New York, 2009.
- W. Pokojski, P. Pokojska, "Voronoi diagrams – inventor, method, applications," Polish Cartographical Review, Vol. 50, no. 3, pp. 141–150, 2018. DOI: 10.2478/pcr-2018-0009.
- A. Thiessen, "Precipitation averages for large areas," Monthly Weather Review, Vol. 39, no. 7, pp. 1082–1089, 1911.
- G. Voronoi, "Nouvelles applications des paramètres continus à la théorie de formes quadratiques," Journal für die Reine und Angewandte Mathematik, Vol. 134, pp. 198–287, 1908.
- S. Wang, J. Rong, Z. Yang, "Transit traffic analysis zone delineating method based on Thiessen polygon," Sustainability, Vol. 6, no. 4, pp. 1821–1832, 2014. DOI: 10.3390/su6041821.
- H. Wu, S. Takahashi, C. Lin, H. Yen, "Voronoi-based label placement for metro maps," 17th International Conference on Information Visualization, London, 2013, pp. 96–101. DOI: 10.1109/IV.2013.11.
- Y. Xu, X. H. Fan, K. G. Liu, L. Shi, B. Xu, F. M. Wang, J. P. Lin, "Applying and practicing of MATLAB programming for Voronoi tessellation," Advanced Materials Research, Vol. 706–708, pp. 391–394, 2013.
- Hacettepe Üniversitesi, "Week 8 Voronoi Diagrams," Hacettepe Üniversitesi, https://web.cs.hacettepe.edu.tr/lectures/lecture7.pdf.

- University of California, Irvine, "Voronoi Diagrams and Their Applications," University of California, Irvine, https://ics.uci.edu/geom/notes/Voronoi1.pdf.
- Brown University, "Introduction to Voronoi Diagrams," Brown University, https://cs.brown.edu/lectures/pdf/notes09.pdf.
- Tufts University, "1 Voronoi Diagrams 2 Definitions," Tufts University, http://www.cs.tufts.edu/comp/voronoi_handout.pdf.
- Purdue University, "Voronoi Diagrams (chapter 7)," Purdue Computer Science, https://www.cs.purdue.edu/homes/slides/vor.pdf.
- RKMVERI (Deemed University), "Voronoi Diagram," RKMVERI, https://cs.rkmvu.ac.in/~sghosh/subhas-lecture.pdf.
- J. D. Boissonnat, "Curved Voronoi diagrams," Archive ouverte HAL, https://hal.science/file/ecg-book-voronoi.pdf, 2007.
- A. Dobrin, "A review of properties and variations of Voronoi diagrams," Whitman College, https://www.whitman.edu/mathematics/dobrinat.pdf.
- Middle East Technical University, "Voronoi Diagrams and Delaunay Triangulation," METU, https://user.ceng.metu.edu.tr/Schedule/week9.pdf.
- University of Illinois Chicago, "Voronoi Diagrams," University of Illinois Chicago, http://homepages.math.uic.edu/~jan/mcs481.pdf.
- C. K. Li, "Voronoi Diagram," Chi-Kwong Li, https://cklixx.people.wm.edu/Voronoi-paper.pdf, 2020.
- P. Felkel, "Voronoi Diagram," CourseWare Wiki, https://cw.fel.cvut.cz/lectures/06-voronoi-split.pdf.
- Massachusetts Institute of Technology, "Lecture 7: Voronoi Diagrams," MIT, http://nms.lcs.mit.edu/~aklmiu/Lecture7.pdf, 2001.
- Instituto de Computação, "ALGORITHMS FOR CONSTRUCTING VORONOI DIAGRAMS," Instituto de Computação, https://www.ic.unicamp.br/~rezende/ensino/Voronoi.pdf.
- The University of New Mexico, "1 Voronoi Diagrams," The University of New Mexico, https://www.cs.unm.edu/~saia/classes/lec/Voronoi.pdf.
- D. M. Yana, "Efficient Computation of Clipped Voronoi Diagram for Optimal Mesh Generation," Microsoft, https://www.microsoft.com/uploads/2016/12/Voronoi.pdf, 2016.

- C. Wormser, "Generalized Voronoi Diagrams and Applications," HAL Thèses, https://theses.hal.science/document.pdf, 2008.
- Purdue University, "Voronoi diagrams and weighted complexes," Purdue University, https://www.cs.purdue.edu/course/Voronoi-CDS.pdf.
- DergiPark, "Path planning of autonomous mobile robots based on Voronoi Diagram and Ant Colony Algorithm," DergiPark, https://dergipark.org.tr/download/article-file.pdf.
- University of Arizona, "Properties of Voronoi Diagrams," University of Arizona, https://www2.cs.arizona.edu/Lecture7.prn.pdf.
- École Normale Supérieure de Lyon, "Convex Hulls, Voronoi Diagrams and Delaunay Triangulations," ENS Lyon, https://www.ens-lyon.fr/Arenaire/JDB-ens-lyon-I.pdf, 2010.
- O. Cheong, "Farthest-polygon Voronoi diagrams," Loria - Inria, https://members.loria.fr/fpvd/slides.pdf.
- E. Friedrich, "The Voronoi diagram in structural optimisation," UCL Discovery, https://discovery.ucl.ac.uk/document.pdf, 2008.
- H. Ledoux, "Computing the 3D Voronoi Diagram Robustly," GDMC, https://gdmc.nl/publications/Computing_3D_Voronoi.pdf, 2007.
- D. Lee, "On k-Nearest Neighbor Voronoi Diagrams in the Euclidean Plane," amedeolucente.it, https://amedeolucente.it/public/VORONOI.pdf, 1982.
- UPC Universitat Politècnica de Catalunya, "STORING THE VORONOI DIAGRAM," UPC, https://dccg.upc.edu/uploads/2020/06/GeoC.pdf.
- M. Mumm, "Voronoi Diagrams," ScholarWorks at University of Montana, https://scholarworks.umt.edu/viewcontent.pdf, 2004.
- M. Erwig, "The Graph Voronoi Diagram with Applications," Oregon State University, https://web.engr.oregonstate.edu/~erwig/papers.pdf.
- Johns Hopkins University, "Voronoi Diagrams and Delaunay Triangulations," Johns Hopkins University, https://www.cs.jhu.edu/~misha/Spring16.pdf.
- UCSB Computer Science, "Voronoi Diagrams," UCSB, https://sites.cs.ucsb.edu/~suri/Voronoi.pdf, 2019.
- Beijing University, "3. Voronoi Diagrams," Beijing University, https://www.math.pku.edu.cn/Note/vmsc.pdf.

- K. E. Hoff III, "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware," CMU School of Computer Science, https://www.cs.cmu.edu/sbp_papers/integrated3.pdf, 2010.
- Bowdoin College, "Voronoi Diagram," Bowdoin College, https://tildesites.bowdoin.edu/Slides/cg-voronoi.pdf.
- University of Florida, "Lecture 7: Voronoi diagrams," University of Florida, https://www.cise.ufl.edu/kreveldmorevoronoi.pdf.
- Columbia University, "Voronoi Path Planning," Department of Computer Science, Columbia University, https://www.cs.columbia.edu/~allen/NOTES.pdf.
- Duke University, "Lecture 11: Voronoi diagram, Delaunay triangulation," Duke University, https://courses.cs.duke.edu/fall05/cps234/notes.pdf, 2005.
- F. Anton, "The Voronoi diagram of circles and its application to the Euclidean Plane," Danmarks Tekniske Universitet - DTU, http://www2.imm.dtu.dk/Anton_etal_2009.pdf.
- Cornell University, "Voronoi diagrams and applications," Cornell Computer Science Department, https://www.cs.cornell.edu/courses/lectures.pdf.
- Calculate, "Voronoi Diagrams Exercises," Calculate, https://calculate.org.au/uploads/sites/2019/01/Voronoi.pdf.
- D. Tubbenhauer, "What is...a Voronoi diagram? Or: Distance diagrammatically," Daniel Tubbenhauer, https://www.dtubbenhauer.com/slides/72-voronoi.pdf.
- Y. Li, "Area Queries Based on Voronoi Diagrams," arXiv, https://arxiv.org/pdf.pdf, 2019.
- E. T. K. Sang, "VORONOI DIAGRAMS WITHOUT BOUNDING BOXES," ResearchGate, https://www.researchgate.net/fulltext/VORONOI.pdf, 2015.
- K. Lu, "HVS: Hierarchical Graph Structure Based on Voronoi," VLDB Endowment, https://www.vldb.org/pvldb/vol15/p246-lu.pdf.
- University of California, Irvine, "Voronoi Diagrams," University of California, Irvine, https://ics.uci.edu/teach/geom/notes/Voronoi.pdf.
- TU Delft, "Tetrahedralisations and 3D Voronoi diagrams," TU Delft 3D Geoinformation, https://3d.bk.tudelft.nl/data/handout2.2.pdf, 2021.
- H. Imai, "Voronoi Diagram in the Laguerre Geometry and Its Applications," Simon Fraser University, https://www2.cs.sfu.ca/~binay/LaguerreGeometry.pdf, 1985.

- TI Education, "Voronoi Activity," TI Education, https://education.ti.com/media.pdf.
- Inria, "Voronoi Diagrams, Delaunay Triangulations and Polytopes," Inria, https://team.inria.fr/files/2016/01/2-delaunay-1.pdf.
- S. Fortune, "27 VORONOI DIAGRAMS AND DELAUNAY," University Northridge, http://www.csun.edu/~ctoth/Handbook/chap27.pdf.
- Universidad de Sevilla, "Lecture 11: Voronoi diagrams," Universidad de Sevilla, https://personal.us.es/almar/11voronoi.pdf.
- Drones and Autonomous Systems Lab, "Introduction to Voronoi Diagrams," DASL, https://www.daslhub.org/wiki/lib/exe/fetch.pdf.
- R. C. Lindenbergh, "Voronoi diagrams," Universiteit Utrecht, https://dspace.library.uu.nl/bitstream/handle.pdf, 2002.
- Rensselaer Polytechnic Institute (RPI), "Lecture 10: Voronoi Diagrams, Part 1," RPI, https://www.cs.rpi.edu/~cutler/classes/lectures.pdf.
- UMD Computer Science, "Lecture 11 Voronoi Diagrams and Fortune's Algorithm," UMD, https://www.cs.umd.edu/class/Lects/lect11-vor.pdf.
- K. Q. Brown, "Voronoi Diagrams from Convex Hull," Department of Mathematics, Washington University in St. Louis, https://www.math.wustl.edu/pmf/KQBrown.pdf, 1979.
- UBC Computer Science Department, "1 Voronoi Diagram," UBC, https://www.cs.ubc.ca/~will/scribe.pdf, 2022.

Chapter 8

Appendix

```python
import random
import numpy as np
import matplotlib.pyplot as plt


def norm(vector):
    return np.sqrt(np.sum(np.square(vector)))


def custom_argsort(arr):
    return sorted(range(len(arr)), key=lambda x: arr[x])


def generate_points(n, r_min, r_max, theta_min, theta_max):
    points = []
    for _ in range(n):
        r = random.uniform(r_min, r_max)
        theta = random.uniform(theta_min, theta_max)
        x = r * np.cos(np.radians(theta))
        y = r * np.sin(np.radians(theta))
        distance = np.sqrt(x**2 + y**2)
        points.append((x, y, distance))
    return points


def find_closest_point(points, P_0, closest_points):
    points_array = np.array([point[:2] for point in points])
    P_0_array = np.array(P_0)

    distances = np.array([norm(point - P_0_array) for point in points_array])
    for point in closest_points:
        if point in points:
            index = points.index(point)
            distances[index] = float("inf")

    closest_point_index = np.argmin(distances)
    return closest_point_index, points[closest_point_index]


def find_unit_vector(P_from, P_to):
    vector = np.array(P_to[:2]) - np.array(P_from[:2])
    vector_norm = norm(vector)
    if vector_norm == 0:
        return np.array([0, 0])
    return vector / vector_norm


def filter_points_by_dot_product(points, base_point, reference_vector):
    remaining_points = []
    for point in points:
        if np.array_equal(point[:2], base_point[:2]):
            continue
        unit_vector = find_unit_vector(base_point, point[:2])
        dot_product = np.dot(reference_vector, unit_vector)
        if dot_product >= 0:
            remaining_points.append(point)
    return np.array(remaining_points)


def find_intersection(midpoint1, normal1, midpoint2, normal2):
    A = np.array([normal1, -normal2]).T
    b = np.array(midpoint2) - np.array(midpoint1)
    if np.linalg.det(A) == 0:
        return None
    intersection = linalg_solve(A, b)
```

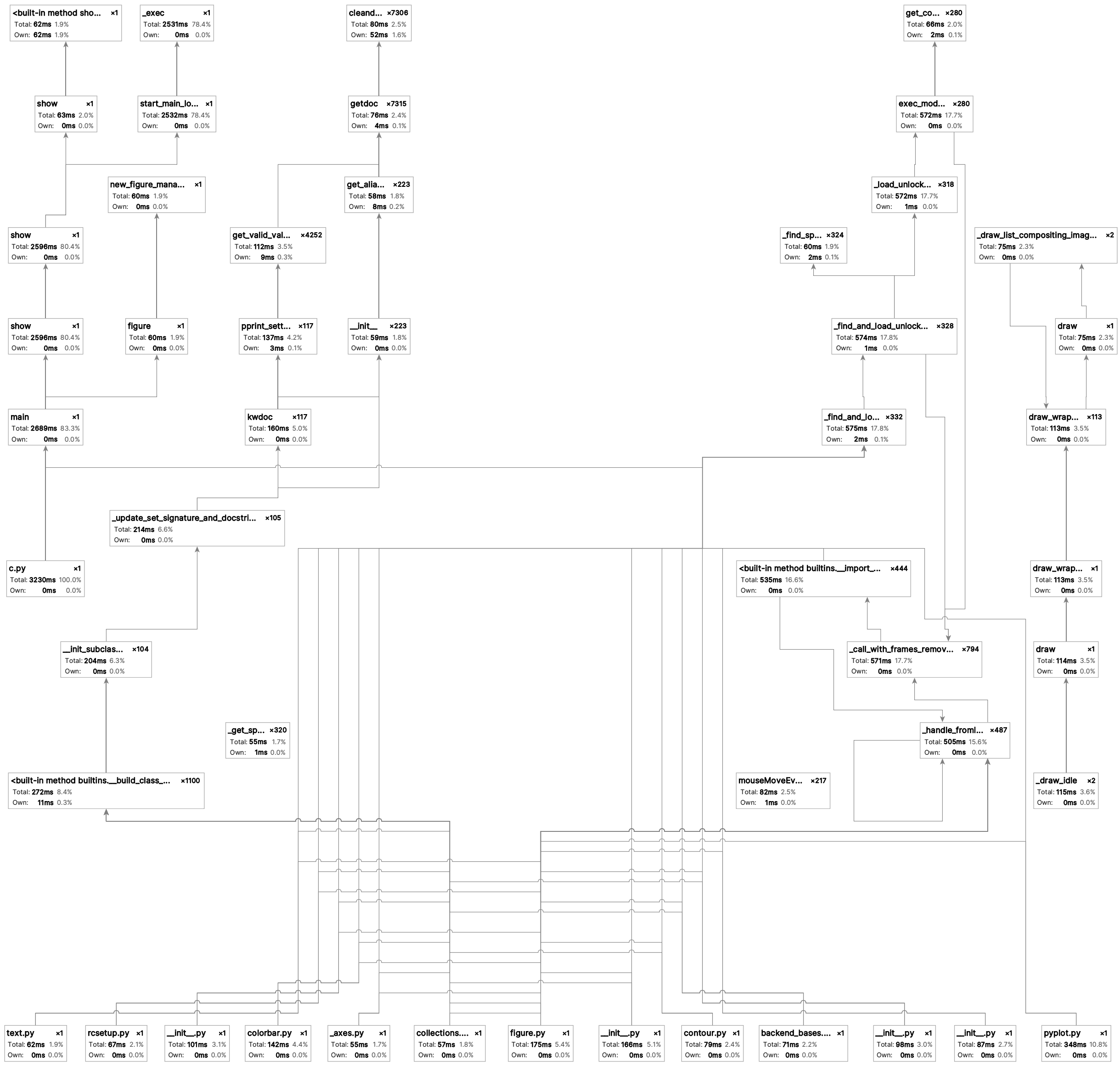
```python
    return midpoint1 + intersection[0] * normal1


def linalg_solve(A, b):
    n = len(A)
    M = [list(row) for row in A]
    for i in range(n):
        M[i].append(b[i])
    for k in range(n):
        max_row = max(range(k, n), key=lambda i: abs(M[i][k]))
        M[k], M[max_row] = M[max_row], M[k]

        for i in range(k + 1, n):
            factor = M[i][k] / M[k][k]
            for j in range(k, n + 1):
                M[i][j] -= factor * M[k][j]

    x = [0] * n
    for i in range(n - 1, -1, -1):
        x[i] = M[i][n] / M[i][i]
        for j in range(i - 1, -1, -1):
            M[j][n] -= M[j][i] * x[i]

    return x


def sort_points_cyclic_order(points):
    center = np.mean(points, axis=0)
    angles = np.arctan2(points[:, 1] - center[1], points[:, 0] - center[0])
    sorted_indices = custom_argsort(angles)
    return points[sorted_indices]


def main():
    P_0 = (0, 0)
    points_first = (
        generate_points(n=5, r_min=2.5, r_max=15, theta_min=5, theta_max=85)
        + generate_points(n=5, r_min=2.5, r_max=15, theta_min=95, theta_max=175)
        + generate_points(n=5, r_min=2.5, r_max=15, theta_min=185, theta_max=265)
        + generate_points(n=5, r_min=2.5, r_max=15, theta_min=275, theta_max=355)
    )
    print(f"All generated points: {points_first}")
    points = points_first.copy()
    closest_points = []

    while len(points) > 0:
        closest_point_index, closest_point = find_closest_point(
            points, P_0, closest_points
        )
        closest_points.append(closest_point)

        reference_vector = find_unit_vector(P_from=closest_point, P_to=P_0)
        points = filter_points_by_dot_product(points, closest_point, reference_vector)

        print(f"Selected closest point: {closest_point}")

        if points.size == 0:
            break

    midpoints = [(np.array(P[:2]) + np.array(P_0)) / 2 for P in closest_points]
    normals = []
    for P in closest_points:
        vector = np.array(P[:2]) - np.array(P_0)
        normal = np.array([-vector[1], vector[0]])
        unit_normal = normal / norm(normal)
```

```
        normals.append(unit_normal)

    midpoints = np.array(midpoints)
    angles = np.arctan2(midpoints[:, 1], midpoints[:, 0])
    sorted_indices = custom_argsort(angles)
    midpoints = midpoints[sorted_indices]
    normals = np.array(normals)[sorted_indices]

    intersection_points = []
    for i in range(len(midpoints)):
        next_index = (i + 1) % len(midpoints)
        intersection = find_intersection(
            midpoints[i], normals[i], midpoints[next_index], normals[next_index]
        )
        if intersection is not None:
            intersection_points.append(intersection)

    plt.figure(figsize=(10, 8))
    plt.scatter(
        [point[0] for point in points_first],
        [point[1] for point in points_first],
        color="gray",
        label="First Generated",
    )
    all_points = closest_points + [P_0]
    x_coords = [point[0] for point in all_points]
    y_coords = [point[1] for point in all_points]
    plt.scatter(x_coords, y_coords, color="green", label="Selected Points")

    for point in closest_points:
        plt.plot([P_0[0], point[0]], [P_0[1], point[1]], "gray", linestyle="dotted")
    plt.scatter(
        [point[0] for point in midpoints],
        [point[1] for point in midpoints],
        color="blue",
        label="Midpoints",
    )

    intersection_points = np.array(intersection_points)
    if intersection_points.size > 0:
        plt.scatter(
            intersection_points[:, 0],
            intersection_points[:, 1],
            color="purple",
            label="Intersections",
        )

        for i in range(len(intersection_points)):
            next_index = (i + 1) % len(intersection_points)
            plt.plot(
                [intersection_points[i][0], intersection_points[next_index][0]],
                [intersection_points[i][1], intersection_points[next_index][1]],
                color="purple",
            )

    plt.xlabel("X Coordinates")
    plt.ylabel("Y Coordinates")
    plt.title("Voronoi Cell and Points")
    plt.grid(True)
    plt.legend()
    plt.show()


if __name__ == "__main__":
    main()
```

```python
from manim import *
import numpy as np
import random

class VoronoiAnimation(Scene):
    def norm(self, vector):
        return np.sqrt(np.sum(np.square(vector)))

    def custom_argsort(self, arr):
        return sorted(range(len(arr)), key=lambda x: arr[x])

    def generate_points(self, n, r_min, r_max, theta_min, theta_max):
        points = []
        for _ in range(n):
            r = random.uniform(r_min, r_max)
            theta = random.uniform(theta_min, theta_max)
            x = r * np.cos(np.radians(theta))
            y = r * np.sin(np.radians(theta))
            points.append(np.array([x, y, 0]))  # Ensure points are 3D
        return points

    def find_closest_point(self, points, P_0, closest_points):
        if len(points) == 0:
            raise ValueError("No points to find the closest point from.")

        distances = np.array([self.norm(point - P_0) for point in points])
        for point in closest_points:
            indices = np.where((points == point).all(axis=1))
            if len(indices[0]) > 0:
                index = indices[0][0]
                distances[index] = float('inf')

        closest_point_index = np.argmin(distances)
        return closest_point_index, points[closest_point_index]

    def find_unit_vector(self, P_from, P_to):
        vector = P_to - P_from
        vector_norm = self.norm(vector)
        if vector_norm == 0:
            return np.array([0, 0, 0])
        return vector / vector_norm

    def filter_points_by_dot_product(self, points, base_point, reference_vector):
        remaining_points = []
        for point in points:
            if np.array_equal(point, base_point):
                continue
            unit_vector = self.find_unit_vector(base_point, point)
            dot_product = np.dot(reference_vector, unit_vector)
            if dot_product >= 0:
                remaining_points.append(point)
        return np.array(remaining_points)

    def find_intersection(self, midpoint1, normal1, midpoint2, normal2):
        A = np.array([normal1, -normal2]).T[:2, :2]
        b = midpoint2[:2] - midpoint1[:2]
        if np.linalg.det(A) == 0:
            return None
        intersection = np.linalg.solve(A, b)
        return midpoint1 + intersection[0] * normal1

    def construct(self):
        P_0 = np.array([0, 0, 0])
        points_first = (
            self.generate_points(n=5, r_min=2.5, r_max=15, theta_min=5, theta_max=85)
```

```
+
            self.generate_points(n=5, r_min=2.5, r_max=15, theta_min=95, theta_max=175
) +
            self.generate_points(n=5, r_min=2.5, r_max=15, theta_min=185, theta_max=26
5) +
            self.generate_points(n=5, r_min=2.5, r_max=15, theta_min=275, theta_max=35
5)
        )
        points = np.array(points_first)
        closest_points = []

        all_points = np.array(points_first + [P_0])
        min_x, min_y, _ = np.min(all_points, axis=0)
        max_x, max_y, _ = np.max(all_points, axis=0)

        scene_width = max_x - min_x
        scene_height = max_y - min_y
        max_dim = max(scene_width, scene_height)

        scale_factor = 6 / max_dim
        all_points = (all_points - np.array([(min_x + max_x) / 2, (min_y + max_y) / 2,
 0])) * scale_factor

        points = all_points[:-1]
        P_0 = all_points[-1]

        new_center = np.array([(min_x + max_x) / 2, (min_y + max_y) / 2, 0]) * scale_f
actor

        axes = Axes().shift(np.append(-new_center[:2], 0))
        origin_dot = Dot(P_0, color=RED)
        self.play(Create(axes), Create(origin_dot))

        initial_dots = [Dot(point, color=GRAY) for point in points]
        self.play(*[Create(dot) for dot in initial_dots])
        self.wait(2)

        while len(points) > 0:
            closest_point_index, closest_point = self.find_closest_point(points, P_0,
closest_points)
            closest_points.append(closest_point)

            reference_vector = self.find_unit_vector(P_from=closest_point, P_to=P_0)
            points = self.filter_points_by_dot_product(points, closest_point, referenc
e_vector)

            self.play(Create(Dot(closest_point, color=GREEN)))
            self.play(Create(Line(P_0, closest_point, color=GRAY, stroke_width=2, stro
ke_opacity=0.5)))

            perp_vector = np.array([-reference_vector[1], reference_vector[0], 0])
            start_point = closest_point - perp_vector * 10
            end_point = closest_point + perp_vector * 10
            perp_line = Line(start_point, end_point, color=YELLOW, stroke_width=2)
            self.play(Create(perp_line))

            remaining_dots = [Dot(point, color=GRAY, fill_opacity=0.3 if point in poin
ts else 0.1) for point in all_points[:-1]]
            animations = [dot.animate.set_fill(opacity=0.1) for dot in remaining_dots
if dot.get_center() not in points]
            if animations:
                self.play(*animations)

            self.wait(1)
```

```
            if points.size == 0:
                break

        midpoints = [(P + P_0) / 2 for P in closest_points]
        normals = []
        for P in closest_points:
            vector = P - P_0
            normal = np.array([-vector[1], vector[0], 0])
            unit_normal = normal / self.norm(normal)
            normals.append(unit_normal)

        midpoints = np.array(midpoints)
        angles = np.arctan2(midpoints[:, 1], midpoints[:, 0])
        sorted_indices = self.custom_argsort(angles)
        midpoints = midpoints[sorted_indices]
        normals = np.array(normals)[sorted_indices]

        for i in range(len(midpoints)):
            start_point = midpoints[i] - normals[i] * 10
            end_point = midpoints[i] + normals[i] * 10
            perp_line = Line(start_point, end_point, color=YELLOW, stroke_width=0.5)
            self.play(Create(perp_line))
            self.wait(0.5)

        intersection_points = []
        for i in range(len(midpoints)):
            next_index = (i + 1) % len(midpoints)
            intersection = self.find_intersection(midpoints[i], normals[i], midpoints[
next_index], normals[next_index])
            if intersection is not None:
                intersection_points.append(intersection)

        self.play(*[Create(Dot(midpoint, color=BLUE)) for midpoint in midpoints])
        self.wait(2)

        intersection_points = np.array(intersection_points)
        if intersection_points.size > 0:
            self.play(*[Create(Dot(intersection, color=PURPLE)) for intersection in in
tersection_points])

            for i in range(len(intersection_points)):
                next_index = (i + 1) % len(intersection_points)
                self.play(Create(Line(intersection_points[i], intersection_points[next
_index], color=PURPLE)))

        self.wait(2)

        if intersection_points.size > 0:
            polygon = Polygon(*intersection_points, color=YELLOW, fill_opacity=0.5)
            self.play(Create(polygon))

        self.wait(2)
```


<built-in method sho... ×1 Total: 62ms 1.9% Own: 62ms 1.9%
_exec ×1 Total: 2531ms 78.4% Own: 0ms 0.0%
cleand... ×7306 Total: 80ms 2.5% Own: 52ms 1.6%
get_co... ×280 Total: 66ms 2.0% Own: 2ms 0.1%
show ×1 Total: 63ms 2.0% Own: 0ms 0.0%
start_main_lo... ×1 Total: 2532ms 78.4% Own: 0ms 0.0%
getdoc ×7315 Total: 78ms 2.4% Own: 4ms 0.1%
exec_mod... ×280 Total: 572ms 17.7% Own: 0ms 0.0%
new_figure_mana... ×1 Total: 60ms 1.9% Own: 0ms 0.0%
get_alia... ×223 Total: 58ms 1.8% Own: 8ms 0.2%
_load_unlock... ×318 Total: 572ms 17.7% Own: 1ms 0.0%
show ×1 Total: 2596ms 80.4% Own: 0ms 0.0%
get_valid_val... ×4252 Total: 112ms 3.5% Own: 9ms 0.3%
_find_sp... ×324 Total: 60ms 1.9% Own: 2ms 0.1%
_draw_list_compositing_imag... ×2 Total: 75ms 2.3% Own: 0ms 0.0%
show ×1 Total: 2596ms 80.4% Own: 0ms 0.0%
figure ×1 Total: 60ms 1.9% Own: 0ms 0.0%
pprint_sett... ×117 Total: 137ms 4.2% Own: 3ms 0.1%
__init__ ×223 Total: 59ms 1.8% Own: 0ms 0.0%
_find_and_load_unlock... ×328 Total: 574ms 17.8% Own: 1ms 0.0%
draw ×1 Total: 75ms 2.3% Own: 0ms 0.0%
main ×1 Total: 2689ms 83.3% Own: 0ms 0.0%
kwdoc ×117 Total: 160ms 5.0% Own: 0ms 0.0%
_find_and_lo... ×332 Total: 575ms 17.8% Own: 2ms 0.1%
draw_wrap... ×113 Total: 113ms 3.5% Own: 0ms 0.0%
_update_set_signature_and_docstri... ×105 Total: 214ms 6.6% Own: 0ms 0.0%
c.py ×1 Total: 3230ms 100.0% Own: 0ms 0.0%
<built-in method builtins.__import_... ×444 Total: 535ms 16.6% Own: 0ms 0.0%
draw_wrap... ×1 Total: 113ms 3.5% Own: 0ms 0.0%
__init_subclas... ×104 Total: 204ms 6.3% Own: 0ms 0.0%
_call_with_frames_remov... ×794 Total: 571ms 17.7% Own: 0ms 0.0%
draw ×1 Total: 114ms 3.5% Own: 0ms 0.0%
_get_sp... ×320 Total: 55ms 1.7% Own: 1ms 0.0%
_handle_froml... ×487 Total: 505ms 15.6% Own: 0ms 0.0%
<built-in method builtins.__build_class_... ×1100 Total: 272ms 8.4% Own: 11ms 0.3%
mouseMoveEv... ×217 Total: 82ms 2.5% Own: 1ms 0.0%
_draw_idle ×2 Total: 115ms 3.6% Own: 0ms 0.0%
text.py ×1 Total: 62ms 1.9% Own: 0ms 0.0%
rcsetup.py ×1 Total: 67ms 2.1% Own: 0ms 0.0%
__init__.py ×1 Total: 101ms 3.1% Own: 0ms 0.0%
colorbar.py ×1 Total: 142ms 4.4% Own: 0ms 0.0%
_axes.py ×1 Total: 55ms 1.7% Own: 0ms 0.0%
collections.... ×1 Total: 57ms 1.8% Own: 0ms 0.0%
figure.py ×1 Total: 175ms 5.4% Own: 0ms 0.0%
__init__.py ×1 Total: 166ms 5.1% Own: 0ms 0.0%
contour.py ×1 Total: 79ms 2.4% Own: 0ms 0.0%
backend_bases.... ×1 Total: 71ms 2.2% Own: 0ms 0.0%
__init__.py ×1 Total: 98ms 3.0% Own: 0ms 0.0%
__init__.py ×1 Total: 87ms 2.7% Own: 0ms 0.0%
pyplot.py ×1 Total: 348ms 10.8% Own: 0ms 0.0%