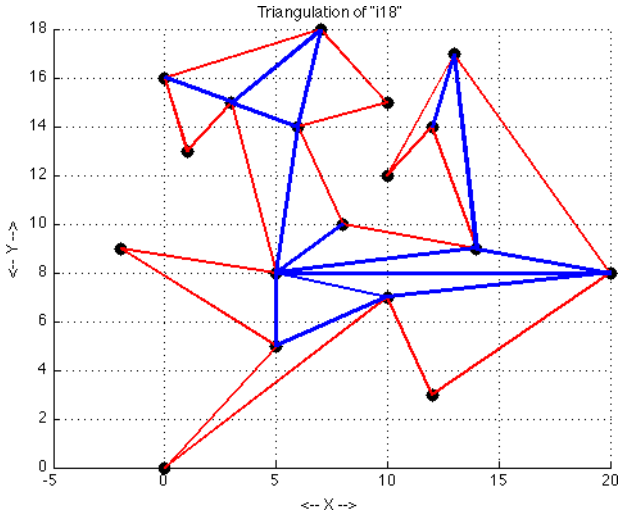
Geometry: Hull, Delaunay, Voronoi
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/math1800_2023/geometry4/geometry4.pdf

---



"Fundamental Geometric Objects"

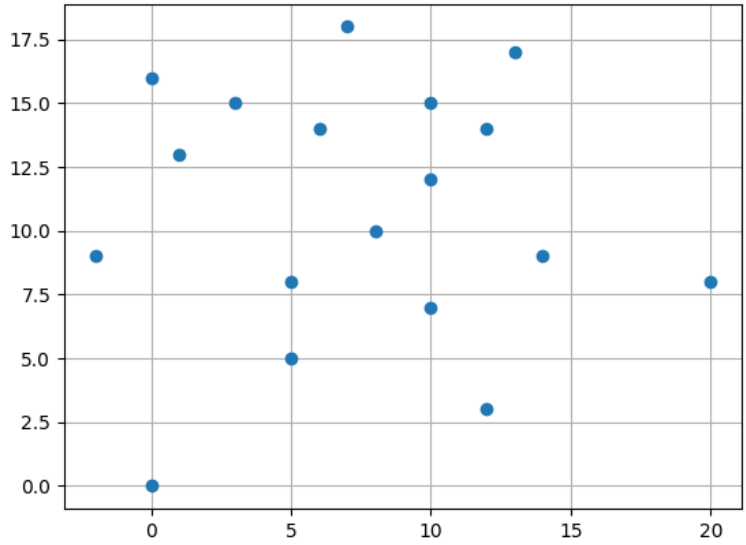
- *Given a set of points,*
  - *the convex hull is a bounding polygon;*
  - *the Delaunay triangulation organizes them into triangles;*
  - *the Voronoi diagram assigns a small area to each point ;*
- *The* `scipy` *library includes functions to compute each of these objects.*

The example programs will all use a common set of 18 data points, which are available on the class web site as *i18.txt*.

1 The convex hull fences in a set of points

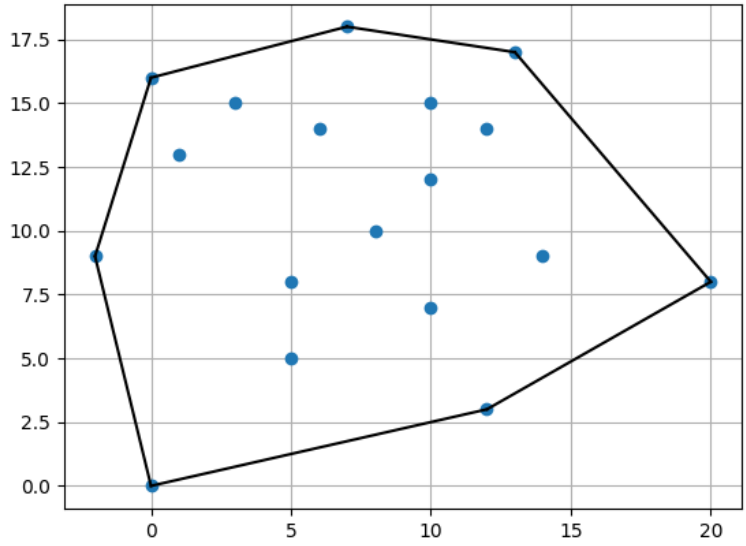
If our data is a scattered set of 2D points, we need some ways to characterize its location and range. The minimum and maximum at least give us a rectangle in which we can be guaranteed to find our data. A more precise bound uses the convex hull.

The convex hull of a set of (2D) points is the smallest convex shape which contains them. Assuming the set is finite, the convex hull is guaranteed to be a polygon, and each vertex of the polygon will be one of the data points. If a convex hull contains two points, it contains all the points on the line segment between them. If it contains three points, it contains all the points in the triangle of which they are the vertices. Thus, the convex hull provides a limited area within which various geometric operations can be carried out.



A sample dataset of 18 points.

If we thought of the data points as nails pounded into a board, then we could imagine a physical way of constructing the convex hull, simply by locating one of the nails that is an outlier, tying a string to it, and then wrapping the string all the way around until we return to our starting point.



The convex hull of 18 points.

Our question then is how to determine the convex hull computationally.

We can always find one point on the convex hull: simply find the point with the minimum x component. If there are several points with the same minimal x component, choose the one with smallest y component. That gets us started, with a data point we will call vertex **H1**.

Now we have to determine the next vertex of the polygon. We have **N**-1 datapoints to choose from. Let's assume we are trying to build the polygon by following the vertices in the counterclockwise direction. So we have **N**-1 possible edges, from **H1** to each of the unused datapoints.

The correct edge **(H1,H2)** *is the unique line through* **H1** *to some vertex* **V** *with the property that all data points lie to the left of it.*

Once this edge is found, move to vertex H2, and search for the vertex V such that all data points lie to the left of the line from H2 to V.

Repeat this operation until you return to H1.

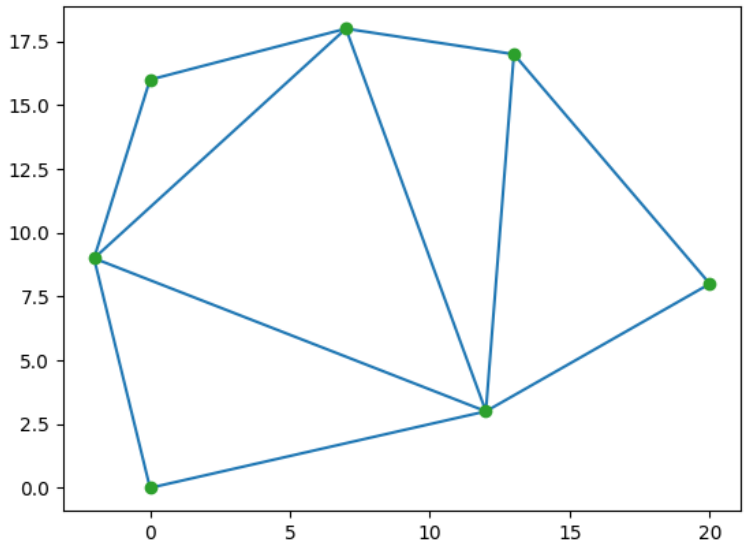
The `scipy.spatial` library includes a function `ConvHull(points)` to compute the convex hull of a set of data stored in `points`. The function returns an object typically denoted `hull`. The `.simplices` component of `hull` is a vector which lists in order the points which constitute the boundary of the convex hull.

```
from scipy.spatial import ConvexHull
import matplotlib.pyplot as plt
import numpy as np

points = np.loadtxt ( 'kn57.txt' )

hull = ConvexHull ( points )

plt.plot ( points[:,0], points[:,1], 'o' )
for simplex in hull.simplices:
  plt.plot ( points[simplex, 0], points[simplex, 1], 'k-' )
```

2 Triangulating a convex hull

A convex hull is a polygon. Every polygon can be neatly cut up into triangles, but for a convex hull, this is especially easy. Suppose the nodes on the boundary are numbered 0 through n-1 in counterclockwise order. Then we can define the following set of n-2 triangles:

```
#    A   B   C
--  ----------
0:   0   1   2
1:   0   2   3
2:   0   3   4
...
n-3: 0 n-2 n-1
```



If the convex hull has been triangulated, then we can easily compute the area of each triangle, and sum them to get the area of the convex hull. This allows us to think about ways of random sampling from the convex hull.

3 Creating a Sampling Probability

Suppose we want to pick a sample point from the convex hull. This point will have to be drawn from one of the subtriangles that make up the hull. How often should we pick from each subtriangle? Surely, if we are sampling uniformly, then we should use the areas of each subtriangle as probability weights. In other words, if one subtriangle is twice as big as the other, we should tend to pick points from it twice as often.

Assume we have n subtriangles, each with area a_i. We want to decide which subtriangle to sample:

1. Define $A = \sum a_i$
2. Define the cumulative density vector `cdf` by $\text{cdf}_i = \sum_{j \le i} a_i / A$
3. Pick random $0 \le r \le 1$.
4. Find i, the index of the smallest cdf_i such that $r \le \text{cdf}_i$
5. Choose a point from subtriangle i;

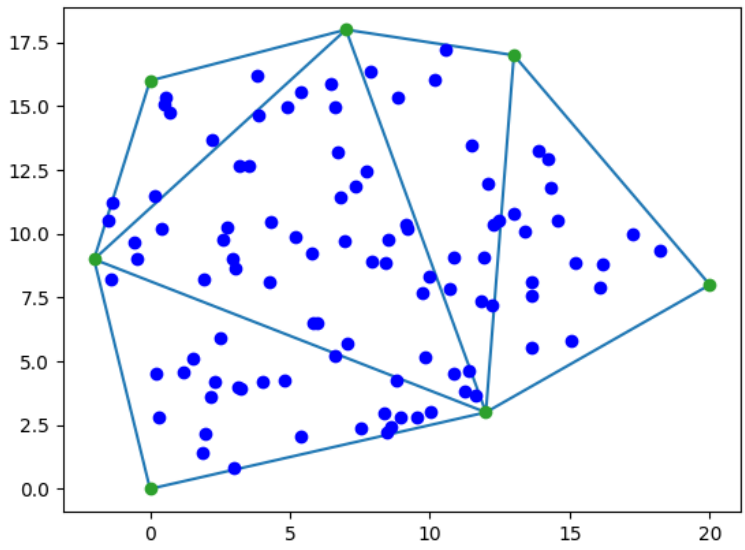
This procedure guarantees that the probability that triangle i is picked is equal to its area relative to the total area.

4 Uniform sampling of a convex hull

In some cases, we can regard our data points as suggesting the limits of variation of our data. However, we may wish to generate new, artificial data points by creating new random points inside the convex hull. We will usually want to do this uniformly, that is, in such a way that regions of equal area have an equal likelihood of being sampled. We can do this by repeating the steps described earlier for sampling a polygon, namely

1. Divide the convex hull into triangles t_i;
2. Pick a random number that selects a triangle i based on `cdf`;
3. Pick two random numbers to sample a point from triangle i.

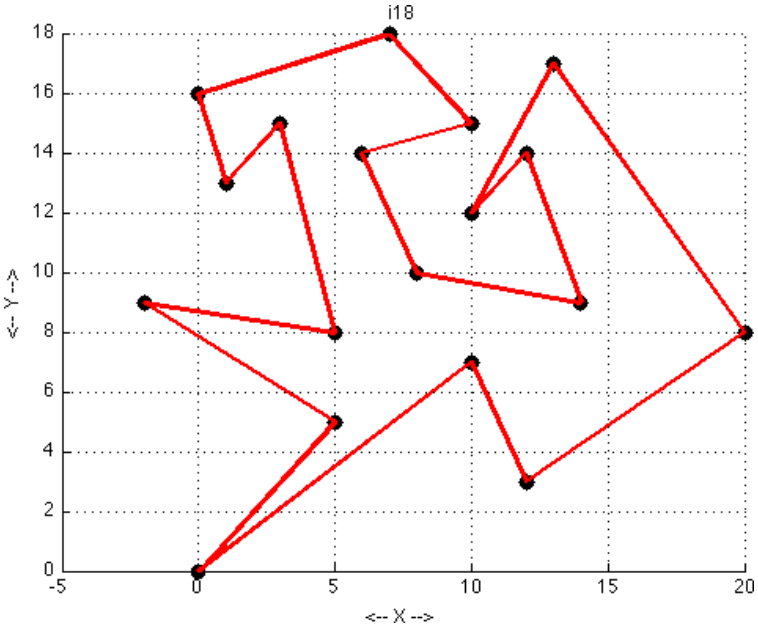
Here is how that procedure works when we sample 100 points from our convex hull.



5 Polygonal triangulation

Now let's consider a different version of the triangulation problem. Instead of receiving a set of points which we are free to bound in a convex hull, we are given a polygon, defined by a sequence of vertices.

Moreover, this polygon might not be convex. If we want to analyze this polygon, we will want to triangulate it, that is, divide it up into triangles. However, the procedure we used for a convex polygon won't work for the general case. Here is how our original data set of points might have been arranged into a very nonconvex polygon:



If we can determine a triangulation of a polygon, then, for instance:

- the area is the sum of the triangle areas;
- a point is in the polygon if it is in one of the triangles;
- the distance to the polygon is the minimum of the distances to any triangle;
- the centroid is the sum of the triangle centroids, weighted by their areas;
- We can estimate integrals over the polygon by summing the estimates over the triangles;

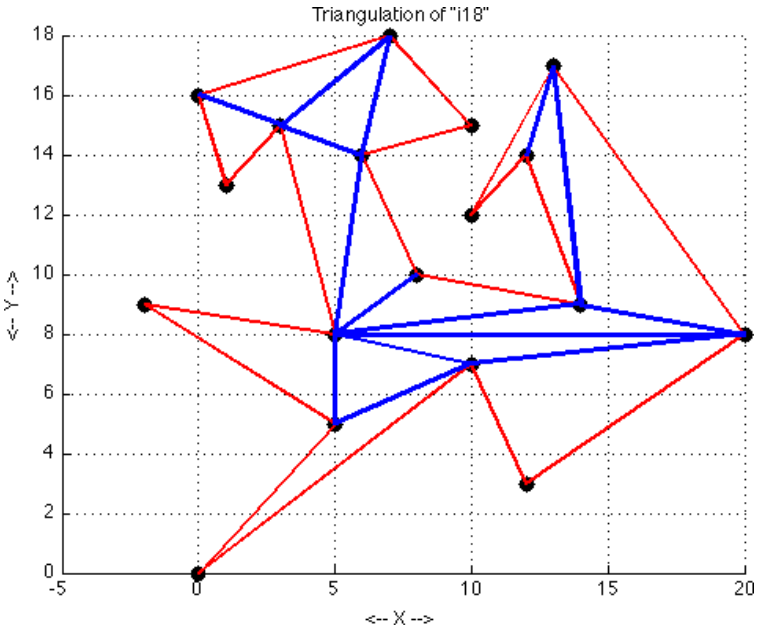
A solution to this problem involves a sort of reverse of mathematical induction. We start with a polygon of a given number of vertices. We know how to triangulate a polygon of 3 vertices (we are done already then!). We don't have a solution for the general problem. But we do know how to reduce the problem to a polygon with one less vertex. And if that's so, we can recursively get down to the 3 vertex case.

How is this possible? An **ear** of a polygon is a triangle formed by three consecutive vertices, in such a way that two edges of the triangle are edges of the polygon, and the third edge is completely contained inside the polygon.

Every simple polygon with at least 4 vertices must have at least two ears, **[Meisters, 1975]**

(A "simple" polygon has no internal holes, and does not have any edges that cross other edges.)

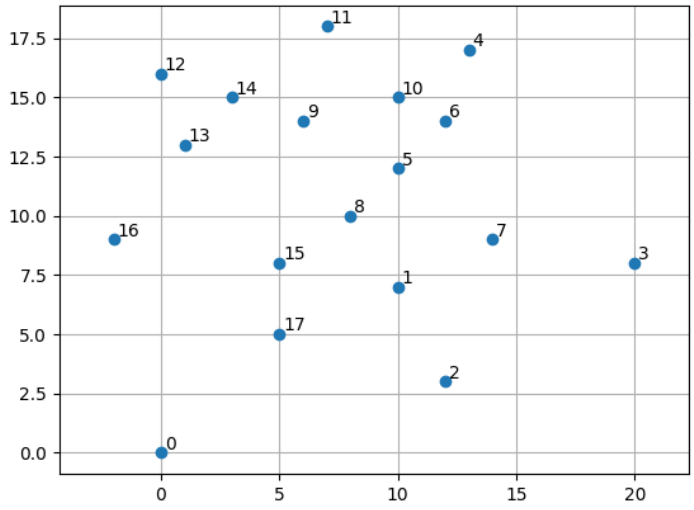
This means that, even if our polygon is complicated, we can slice off one triangular ear at a time, until we get down to the last triangle. And we have found our triangulation!



So whether we start with a set of points, or with a polygon, we can use our knowledge of triangles to answer some important geometric questions.

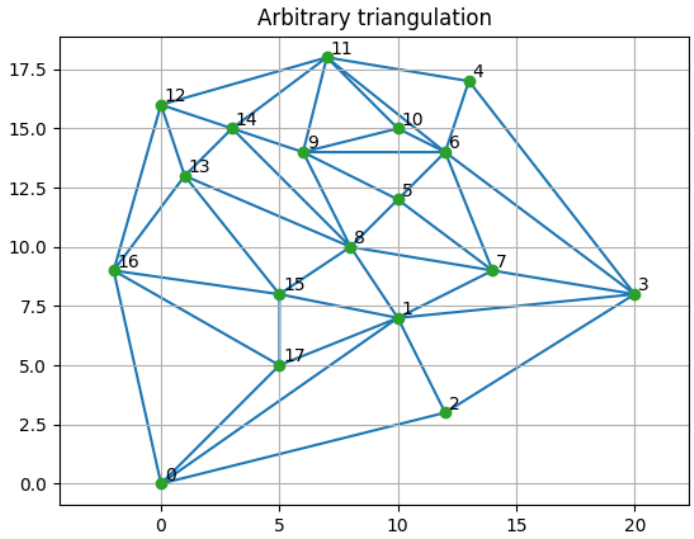
6 Delaunay triangulation

Let us start over with our set of data points in the plane. Instead of fencing them in, now we are looking at connecting pairs of points to form a network. If we make as many connections as possible, without crossing any lines, we will end up with a web of triangles. There are many ways to triangulate this data, but geometrically, it is interesting to do so in a way that creates a smooth, regular pattern.



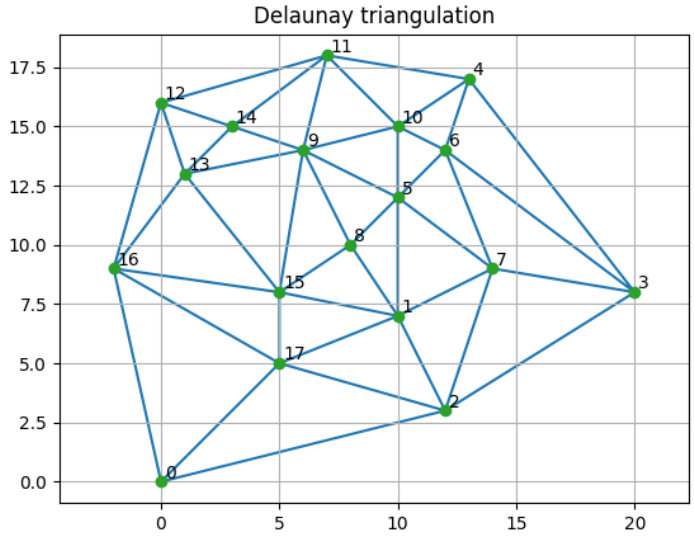
So now we might think of each of our data points as a city, and our job is to connect cities with roads. We aren't allowed to build roads that cross each other. On the other hand, we want to build as many roads as possible. This will result in what is known as a maximal triangulation of our set of points.

As it turns out, the maximal triangulation of a set of points is not unique. There are many ways to do this. Given so many choices, we may want to pick a triangulation based on some characteristic that we can measure. Let's first take a look at a random triangulation of our points to see what happens when we don't care how we connect the data points:



This triangulation is not very attractive. Some of the triangles are long and skinny, with a sharp angle. It turns out that these small angles are a symptom that indicates that this triangulation would not be good for various mathematical tasks.

If we focus on avoiding small angles in our triangles, then the best triangular mesh comes from the Delaunay triangulation. The details of this algorithm are complicated, so let's focus on the triangulation of our data that results:



Algorithms for computing a Delaunay triangulation are too involved for us to describe here. Instead, we will simply point to the fuhnction in `scipy.spatial` function called `Delaunay(points)` which accepts a set of data points and returns an object `tri` with the triangulation information.

```
points = np.loadtxt ( 'i18.txt' )
tri = Delaunay ( points )
plt.triplot ( points[:,0], points[:,1], tri.simplices )
plt.show ( )
```

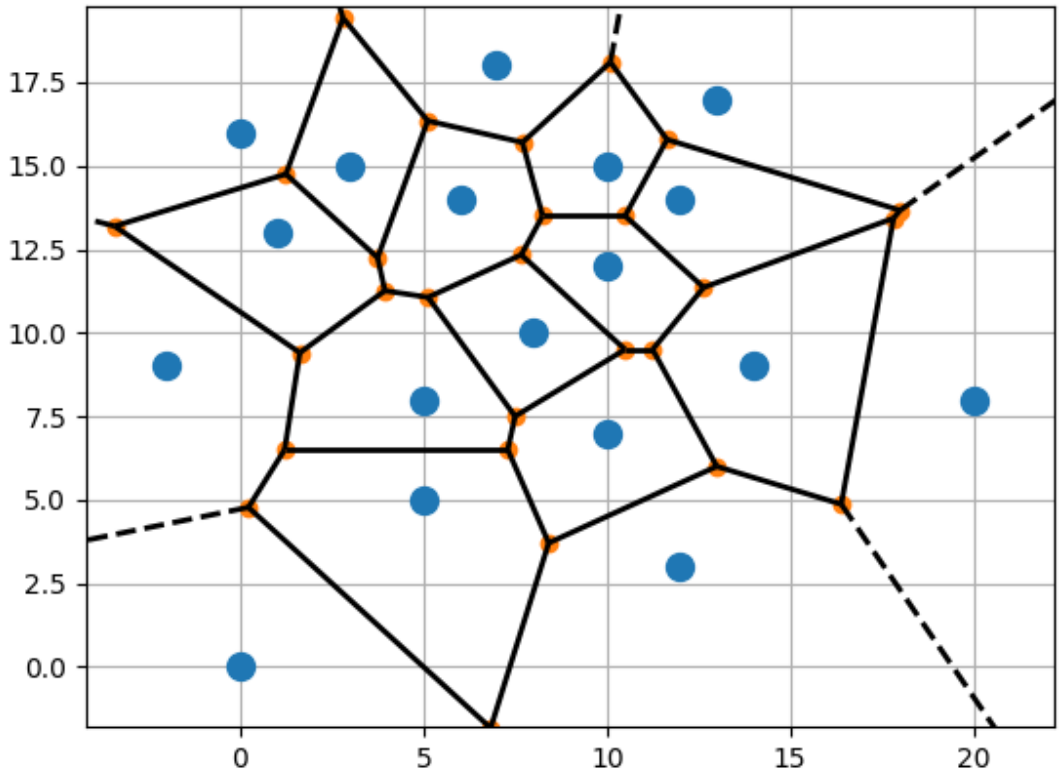
7 The Voronoi diagram

If we again think of our data points as little cities, then another task we might want to carry out would be similar to regarding each city as the capital of a county. Now we need to decide which portion of the region belongs to each city.

The natural rule to use is to assign each point to the nearest city. If we use this rule to organize the region, then we have created a Voronoi diagram for our points. The Voronoi diagram is a very useful structure for computations.

The `scipy.spatial` function `diagram = Voronoi(points)` returns a data structure defining the boundaries of the "counties" surrounding each city. Drawing these subregions is complicated, particularly because some of the subregions are unbounded. Hence, a second function is supplied to create a plot, called `voronoi_plot_2d(diagram )`

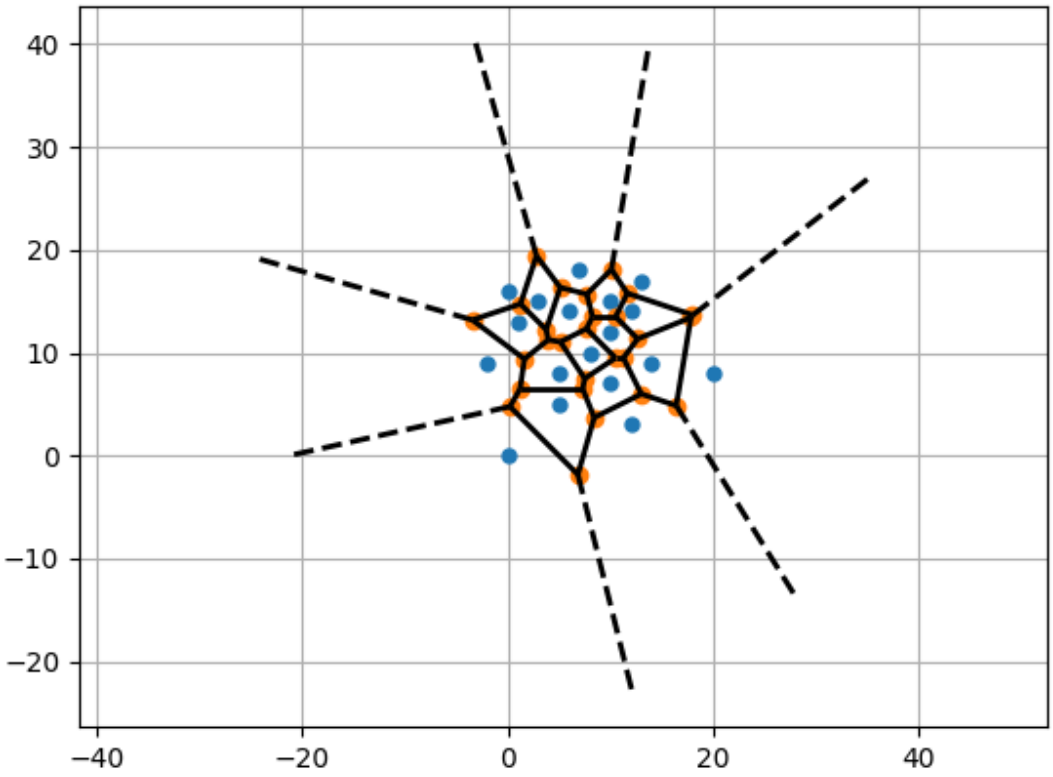
```
points = np.loadtxt ( 'i18.txt' )
diagram = Voronoi ( points )
voronoi_plot_2d ( diagram, line_width = 2, point_size = 20 )
```



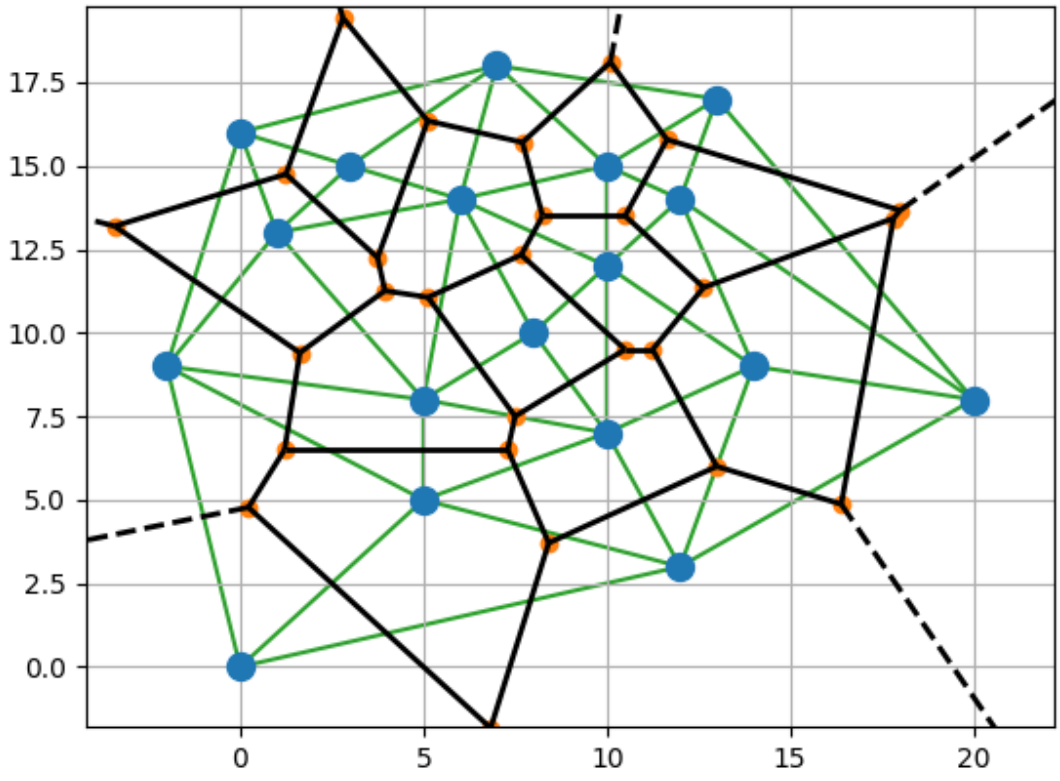
It is worth noting some properties of the Voronoi diagram, as suggested by this plot.

- The diagram is surrounded by a set of unbounded subregions;
- Each subregion is convex;
- When two cities share a border, the line between the cities is perpendicular to the border;

If we take a larger view of the diagram, we can see how the borders of the unbounded regions go off to infinity.



If we include the Delaunay triangulation, we see a network of roads and counties, with the roads crossing the county borders at a perpendicular angle.


17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
5
10
15
20