ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

Name: Foteini Tsavo

Registration Number: 1115201500206

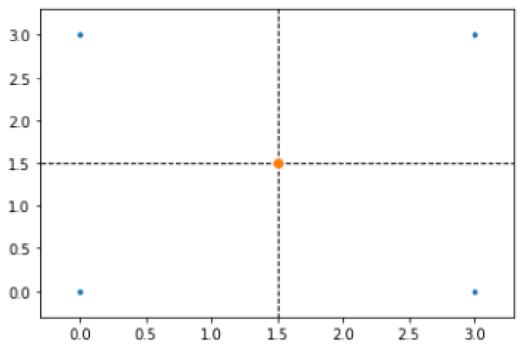
Email: sdi1500206@di.uoa.gr

Computational Geometry

Exercise 1: Compute Voronoi diagrams of different sets of vertices of your choice using the routine Voronoi (and its companion voronoi plot 2d for visualization) from the module scipy.spatial. Plot your results.

A Voronoi Diagram or a Dirichlet tessellation, is the partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other. In this exercise our goal was to create Voronoi diagrams with the help of spicy.spatial, importing Voronoi and voronoi_plot_2d. The first is used to generate a Voronoi diagram that will contain the vertices, regions, ridge points (the ridges are perpendicular between lines drawn between the following input points), ridge vertexes (forming each Voronoi ridge) and point region,which is the index for each Voronoi input point. The second is used to plot the results after the first is applied.

Explaining the code:



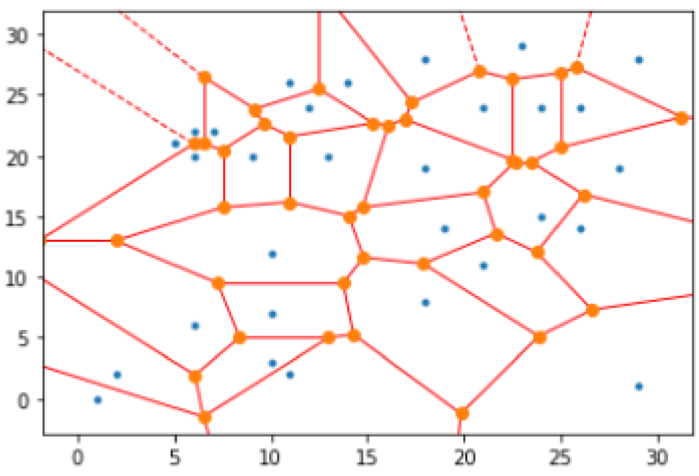
Firstly, we have tried to give as an input to our function an array, where four points of the Euclidean space, part of a convex hull are included, (0,3), (0,0), (3,0), (3,3).

Our results are plotted in a diagram, and the input points are represented in blue color , whereas the orange point is the vertex of the diagram. We have also printed the regions, and a value we see too often is the value -1. This means that one vertex of the digram is not includes in this particular implementation. The array of regions is the one that represent the four regions that are created, each one having a certain number.

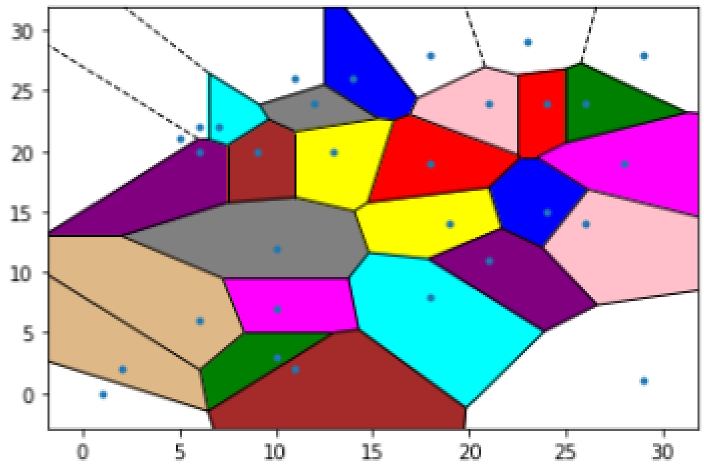
Secondly, we tried to construct the Voronoi diagram of a handful of randomly generated points in the two dimensional space. The number of points that will be generated is from 4 to 80. For the current example, after we ran the program, the points we have generated were:

```
[[24 15]
 [10  3]
 [24 24]
 [10 12]
 [28 19]
 [19 14]
 [ 6 20]
 [ 2  2]
 [11 26]
 [ 5 21]
 [18  8]
 [26 14]
 [ 9 20]
 [14 26]
 [26 24]
 [29  1]
 [18 19]
 [18 28]
 [12 24]
 [ 6 22]
 [10  7]
 [13 20]
 [21 11]
 [ 6  6]
 [23 29]
 [ 7 22]
 [ 1  0]
 [29 28]
 [21 24]
 [11  2]]
```

And their Voronoi tessellation was the following one:

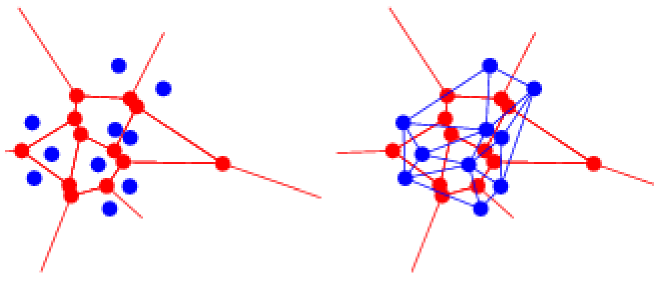


We have also, conducted an experiment, in order to color the Voronoi cells. Practically, we n eed to color the voronoi regions, and that something that is easy to know from the functions that the module provides us with. We first find the regions that each point belongs to, for ins tance the first point (24,15) belongs to region 7. After that, we distinguish the regions and w e take the vertexes for each of them. We color the regions that do not contain the value -1, t herefore we mean that the region are color if they have all their vertexes presented in the di agram.
The results are presented below. We pick colors from a small list that we have create, which contains color names. After we color a region we pop that color from the list and continue. If the color list is empty we refill it.



Exercise 2: Using the routine Delaunay in the module scipy.spatial compute the Delaunay tria ngulation of different sets of vertices of your choice and plot your results.

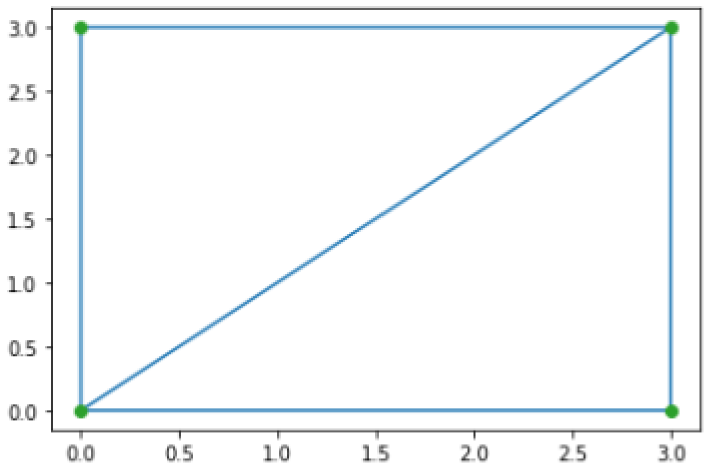
The Delaunay triangulation is a triangulation which is equivalent to the nerve of th e cells in a Voronoi diagram, i.e., that triangulation of the convex hull of the points in the diag ram in which every circumcircle of a triangle is an empty circle. The Delaunay triangulation a nd Voronoi diagram in $\mathbb{R}^2$ are dual to each other.



Voronoi diagram *Delaunay and Voronoi*

Explaining the code:

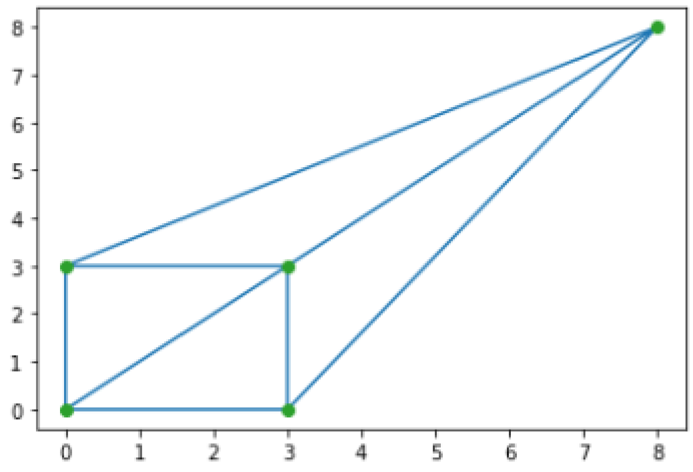
We have started with collinear points, in order to state that with this dataset we cannot find any triangulation. So our next step was to give again the two example dataset we had presen ted in the very first exercise.
We used the example of the four points, and the result was the creation of two triangles.



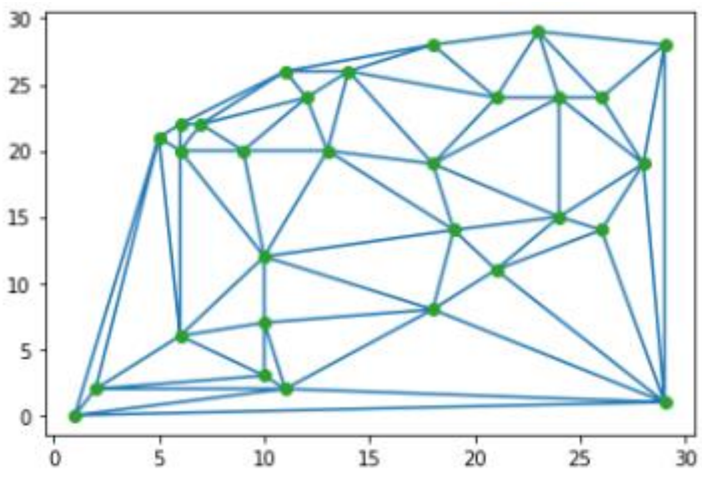
Right after this experiment we tried to add a point to the current dataset, and with this we w ere left with the triangulation presented below:

```
array([[0, 3],
       [0, 0],
       [3, 0],
       [3, 3],
       [8, 8]])
```

```
delaunay_(new_plist)
```



Finally, the points that we had generated above, randomly, were now used to construct a tri angulation.

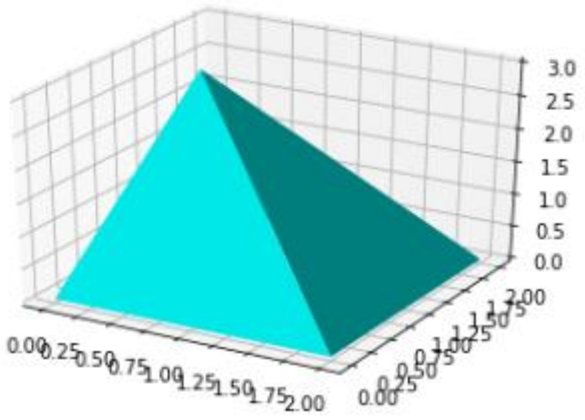


We had also tried this routine for points in the three dimensional space, first with dataset:

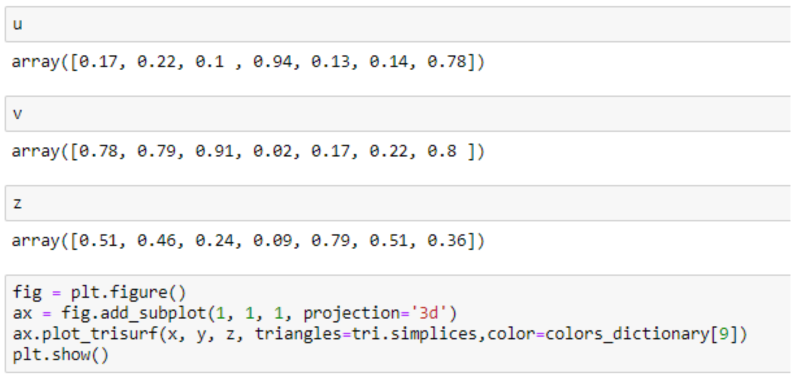
u = np.array([0,0,0.5,2,2])

v = np.array([0,2,1,0,2]

z = np.array([0,0,3,0,0])



Furthermore, we tried with random point generation. A random object was constructed.



```
u
```

```
array([0.17, 0.22, 0.1 , 0.94, 0.13, 0.14, 0.78])
```

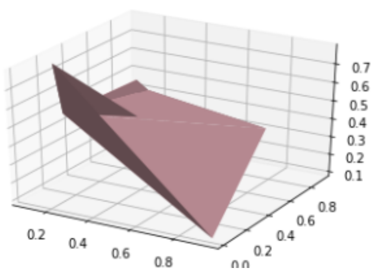
```
v
```

```
array([0.78, 0.79, 0.91, 0.02, 0.17, 0.22, 0.8 ])
```

```
z
```

```
array([0.51, 0.46, 0.24, 0.09, 0.79, 0.51, 0.36])
```

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.simplices,color=colors_dictionary[9])
plt.show()
```



Exercise 3: Compute the shortest path of different set of vertices of your choice in a triangulation. By a path in this setting, we mean a chain of edges of this triangulation. Use the methods in the package scipy.sparse.csgraph.

Our purpose here is to perform a shortest-path graph search on a positive undirected graph. To achieve this we will use the shortest_path routine from spicy.sparce.graph.

Explaining the code:

To start with, the dataset we had given is already a triangle. We find the distances from each point to its neighbors and create with our code a matrix that contains the distances. After that, we apply shortest_path search using **'FW', Floyd-Warshall algorithm. Computational cost is** approximately O$[n^3]$. The input csgraph will be converted to a dense representation. The results are present with the help of a function that gets the path properly. We keep in mind that the matrix Distances represents the distances from the points and the Pred matrix is the one that contains the predecessors. The search stops when we encounter the value of -9999 in the predecessors matrix, which means that we have reached our point. The function requires the matrix of predecessors, the node we will start searching for as well the one that we desire to arrive. In order to translate positions to points, we have parsed as a parameter the list that was inserted as an input. In this way we have the path from point to point printed out.

To continue with, for the same dataset we had used other algorithms too, that have given us the same results.

- 'D' – Dijkstra's algorithm with Fibonacci heaps. Computational cost is approximately O[N(N*k + N*log(N))], where k is the average number of connected edges per node. The input csgraph will be converted to a csr representation.
- 'BF' – Bellman-Ford algorithm. This algorithm can be used when weights are negative. If a negative cycle is encountered, an error will be raised. Computational cost is approximately O[N(N^2k)], where k is the average number of connected edges per node. The input csgraph will be converted to a csr representation.
- 'J' – Johnson's algorithm. Like the Bellman-Ford algorithm, Johnson's algorithm is designed for use when the weights are negative. It combines the Bellman-Ford algorithm with Dijkstra's algorithm for faster computation.

We have done the same experiment for the primary dataset of (0,3),(0,0),(3,0),(3,3).

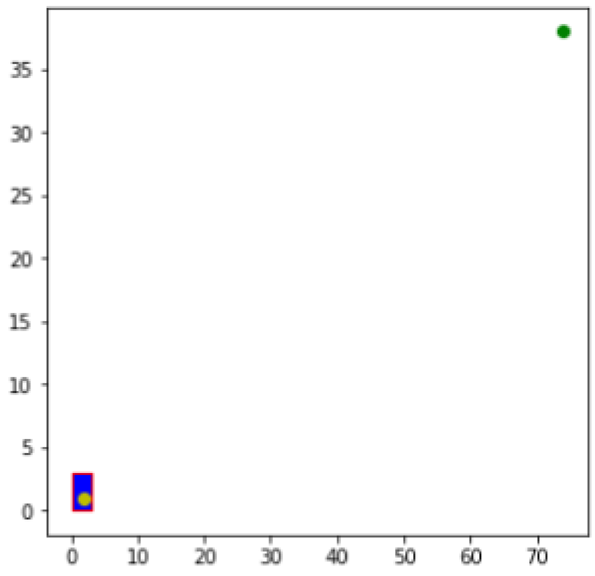
Exercise 4: Experiment yourself with the .encloses point and .encloses methods of the symp y.geometry module usingf polygons or circles to check if they contain certain points of your c hoice. Do the same with contains point or contains points from the Path class from the librar ies of matplotlib.path.

Explain the code:

Start with the dataset of four points that we always give as an input. Each point, or else each list of coordinates, is given as a distinct point for a polygon, using sympy. The first experiment, is to see if the point (2,1) is inside this polygon. The encloses_point function is p ositive to that. Moreover, we try the same thing but now using matplotlib.path and the funct ion contains_point. We get the same result, a positive one. In order to verify this we have pl

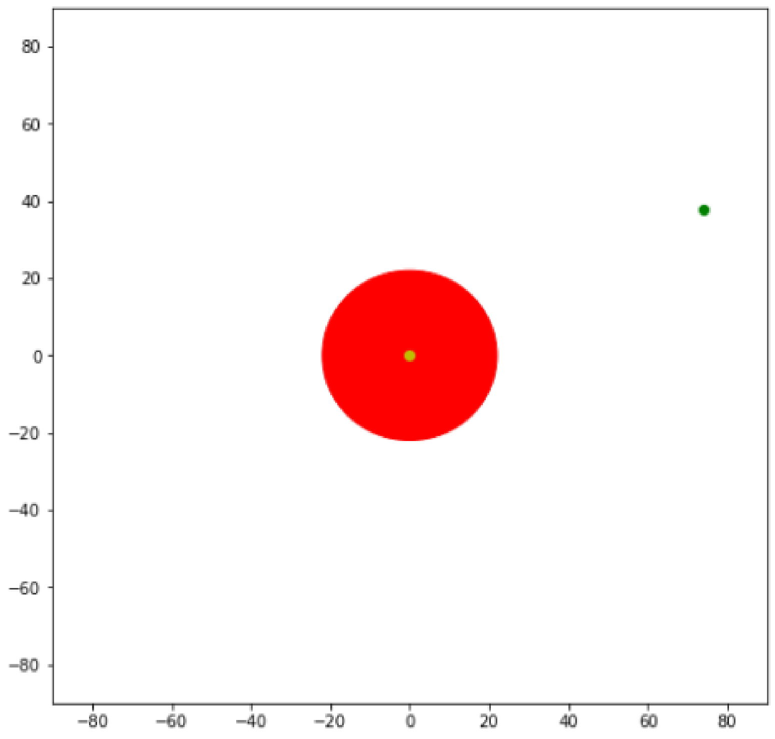
otted the rectangle and its are in blue color and the point that is includes in yellow color. For the same dataset we have tried to search for another point , this time the point is generated randomly. In our case, the generation created the point with coordinates:
Point to be searched is : (74 , 38).

As one can imagine this point is really far from our dataset, so there is no way the function will give us "True" as an answer. And as we have predicted, they did not, the answe r was "False". The green point represent the point that is really far from the polygon we hav e.



Furthermore, we have conducted a pretty similar experiment, but this time using a circle, searching for the same random point and for its origin. As it is expected, when its ori gin was searched the results were positive from both functions. The results of the experimen t are plotted below:

radius = 22



Finally, we have tried searching the random generated point above in the points d ataset, which includes the points that we had generated in the very first exercise.

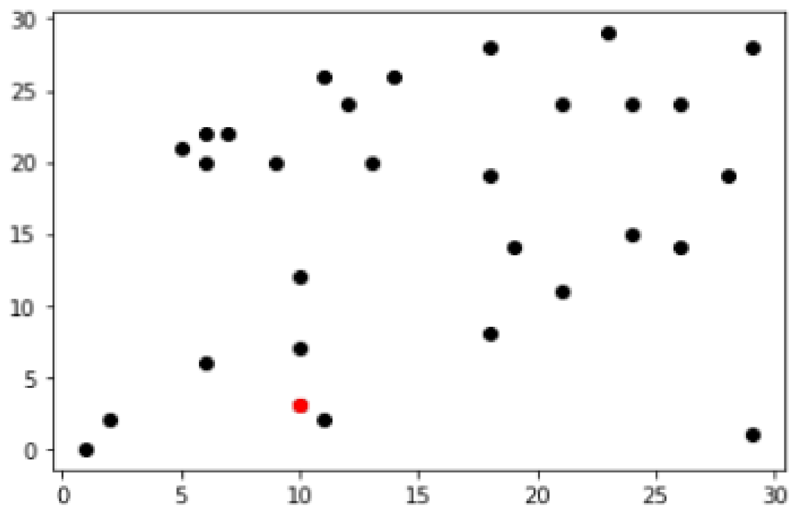
Exercise 5: The problem of finding the Voronoi cell that contains a given location is equivalen t to the search for the nearest neighbor. We can always perform this search with a brute for ce algorithm, but in general there are more elegant and less complex approaches to this pro blem like the kd-trees. In the scipy use the class KDTree to perform some experiments of you r choice.

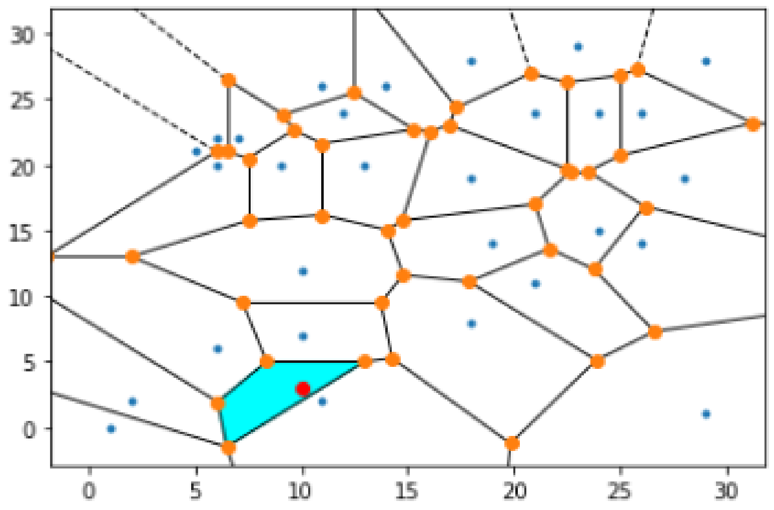
The *k*-d tree is a binary tree in which *every* leaf node is a *k*-dimensional point. Ever y non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. It is a space-partitioning data structure for or ganizing points in a *k*-dimensional space.

In our case, we would like to search the nearest neighbor of a given point and mor eover if we were in a Voronoi diagram, to find the cell that the new point we are searching f or fits best.

Explaining the code:

In this exercise we started with the dataset of the random points that we have bee n using since Exercise 1. The first search we will try, is with point that is already part of the d ataset. We take one of the 4 three points that are included because we are sure that we will not go out bounds due to random generation and list changing number of elements. As we c an predict the query of the nearest neighbor will give us the distance that will be 0 and the r egion that the point belongs to.
Below, we have the representation of point in 2D space and we have colored the point to be search with red color.

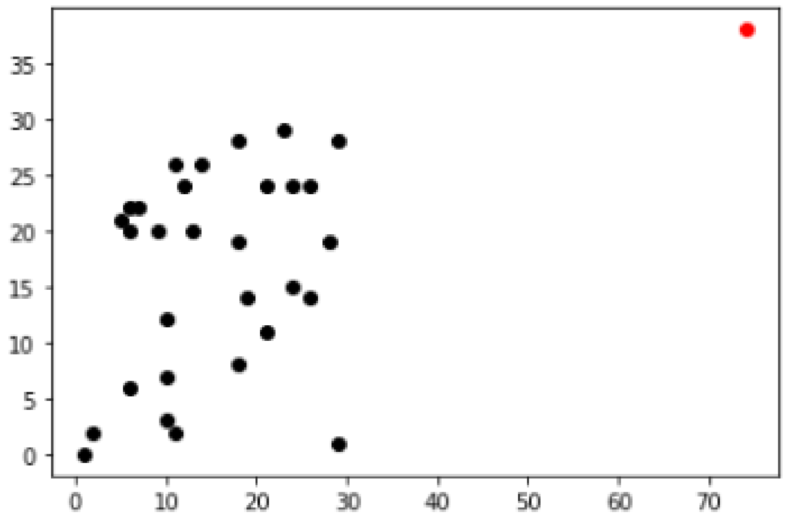




As a result the nearest neighbor for the first query is the same point of Voronoi cel l.

Query: (10,3) → Nearest neighbor: (10, 3)

Secondly, we will try for the same dataset of points, finding the nearest neighbor o f a point that is not included, (74 ,38) .



The distance from the neighbor of the above query seems pretty great with a valu e of 46.09772228646444. the region that it belongs to is 27. So if we remember the very first Voronoi diagram that we had constructed for the same set of points, in region 27 was the po int with position 9.

```
pr = vor2.point_region
pr #Indices of the Voronoi vertices forming each Voronoi region. -1 indicates vertex outside the Voronoi diagram.
```

```
array([ 7,  3, 14, 15, 12,  6, 26,  0, 29, 27,  8,  9, 24, 22, 11, 10, 20,
       18, 21, 25,  2, 23,  5, 16, 17, 28,  1, 13, 19,  4], dtype=int64)
```

As a result we have the following:

Query: (74 ,38) → Nearest neighbor: (5 21)

For the plotting of Voronoi diagrams with a painted region, we have created a fun ction that helps fill the region with color, but in case the region contains -1, one vertex is not on the graph, it prints a warning message and the cell will not get color. This does not mean that the nearest neighbor does not exist but we just can not see it in the plot.