Υπολογιστική Γεωμετρία
Τρίτη Εργασία

Σιώρος Βασίλειος - 1115201500144
Ανδρινοπούλου Χριστίνα - 1115201500006

Ιούνιος 2020

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.

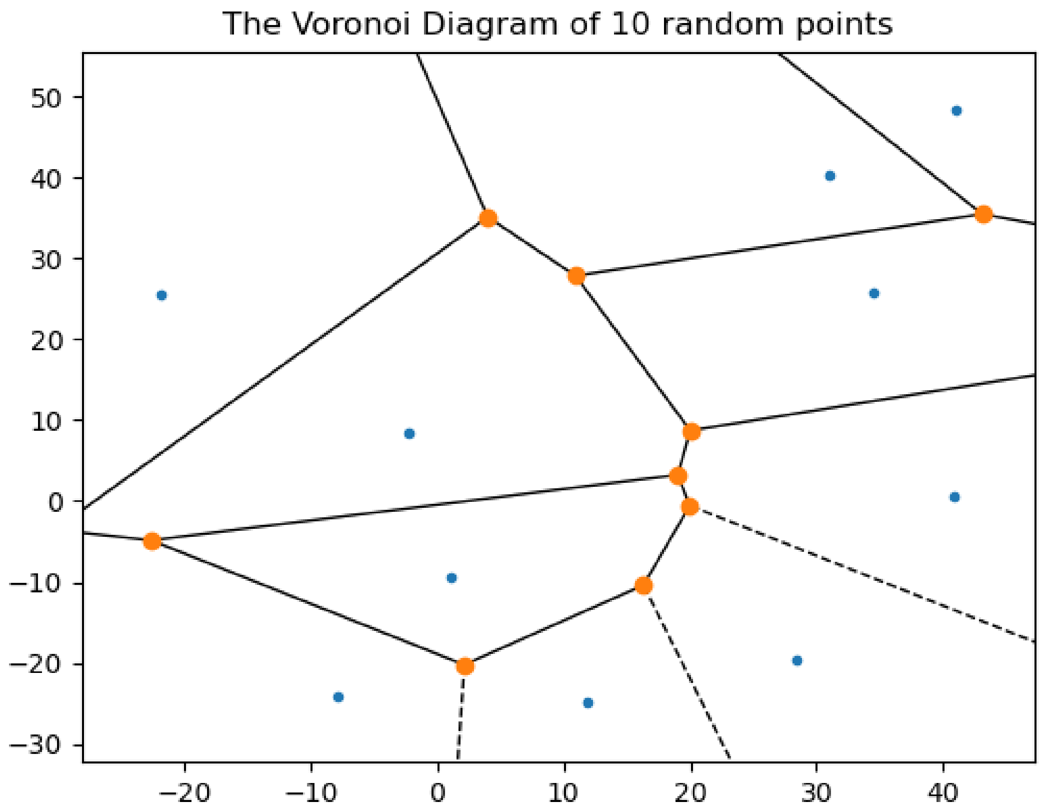

Figure 1: The Voronoi diagram of 10 random points with a random seed of 0

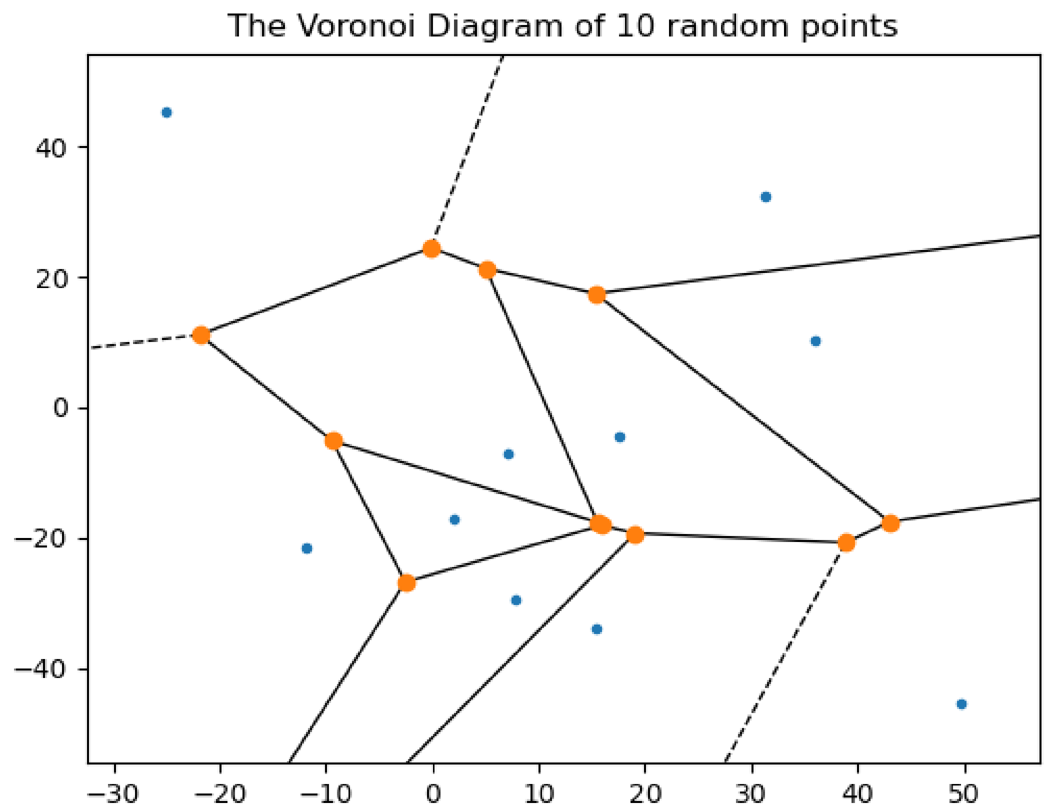

Figure 2: The Voronoi diagram of 10 random points with a random seed of 10

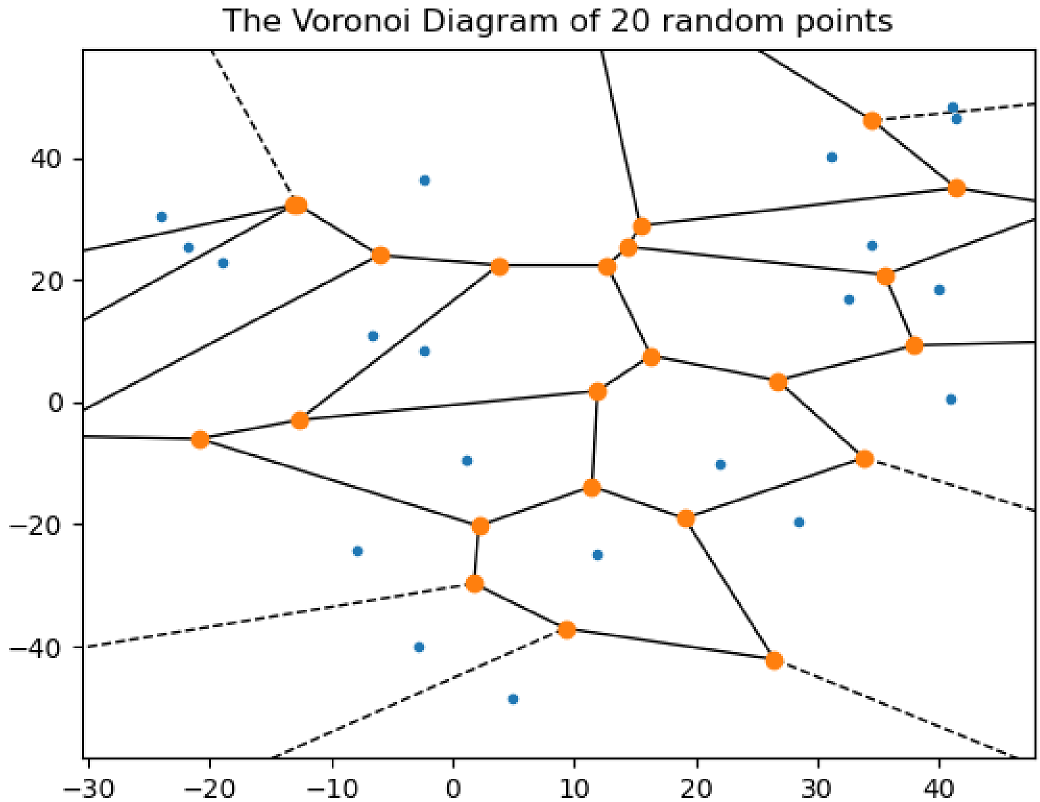

Figure 3: The Voronoi diagram of 20 random points with a random seed of 0

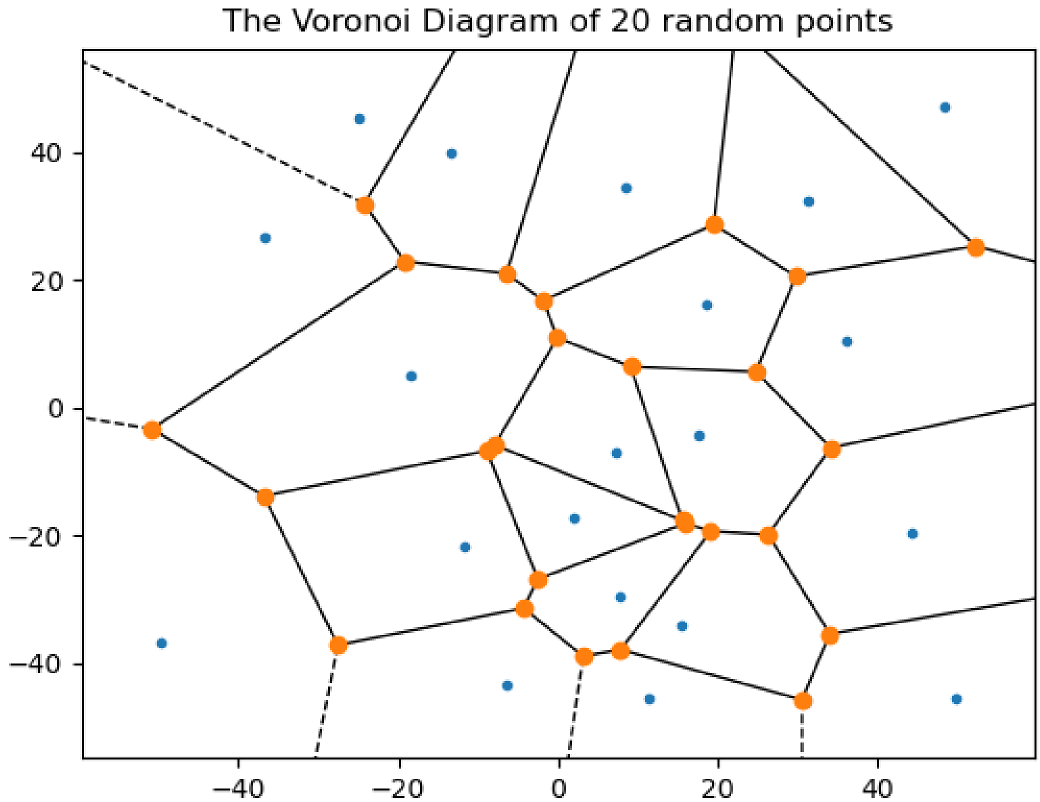

Figure 4: The Voronoi diagram of 20 random points with a random seed of 10

```
$ python -m pip install virtualenv
$ python -m virtualenv .env
$ . .env/Scripts/activate
$ pip install requirements.txt
$ python voronoi.py --help
Usage: voronoi.py [OPTIONS]

  Compute Voronoi diagrams of different sets of vertices

Options:
  -s, --seed INTEGER              the seed of the random number generator
  -n, --number INTEGER            the number of random points to be generated
                                  [default: 10]

  -x, --x-axis <FLOAT FLOAT>...  the minimum and maximum horizontal coordinate
                                  value  [default: -50.0, 50.0]

  -y, --y-axis <FLOAT FLOAT>...  the minimum and maximum vertical coordinate
                                  value  [default: -50.0, 50.0]

  -f, --filename FILENAME         optionally save the figure in PNG format
  --help                          Show this message and exit.
```

Figure 5: How to run the code

2. Using the routine *Delaunay* in the module scipy.spatial compute the Delaunay triangulation of different sets of vertices of your choice and plot your results.

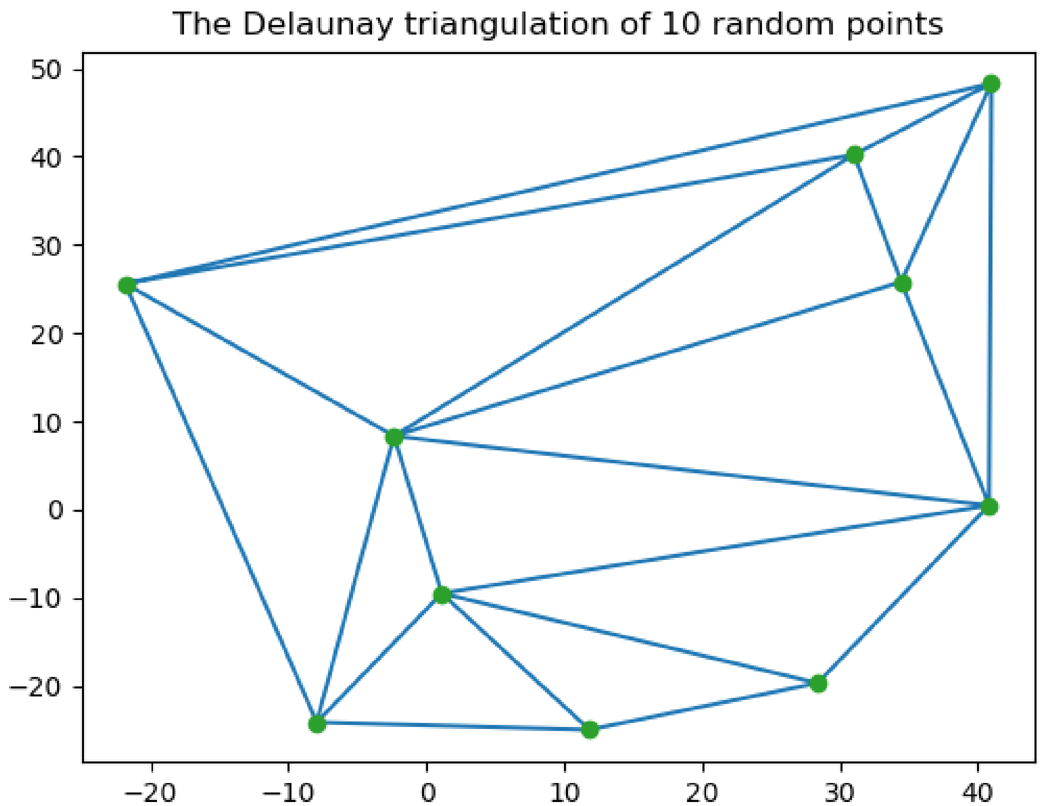

Figure 6: The Delaunay triangulation of 10 random points with a random seed of 0

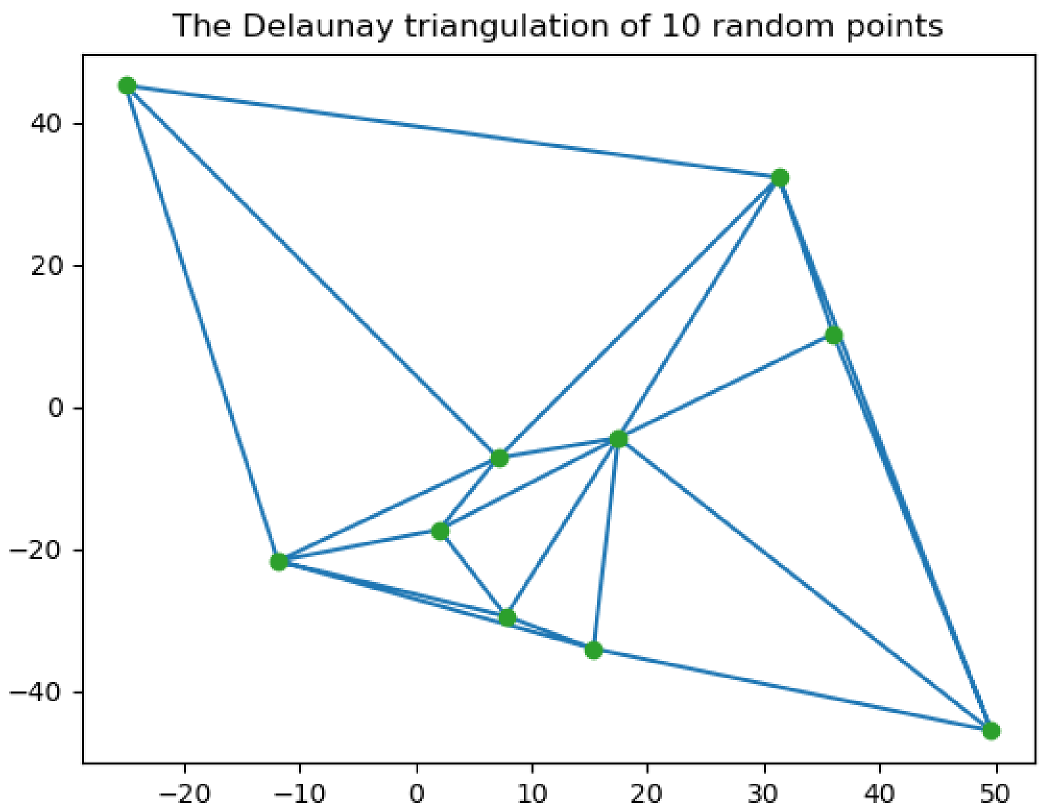

Figure 7: The Delaunay triangulation of 10 random points with a random seed of 10

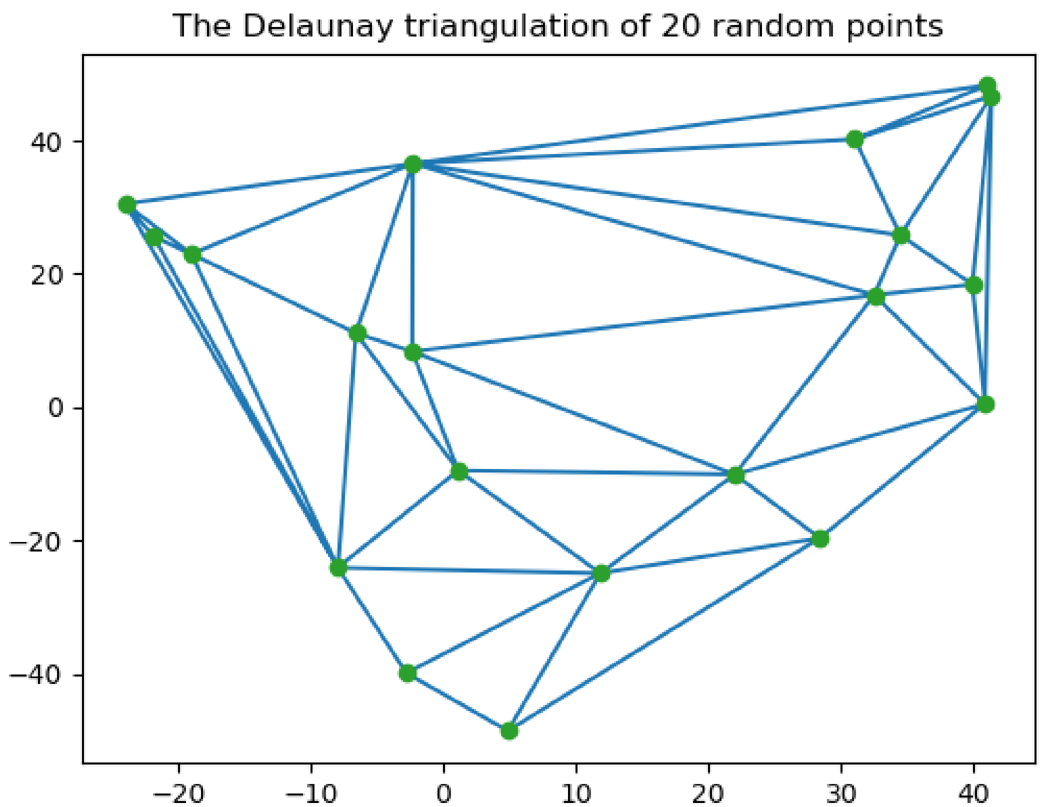

Figure 8: The Delaunay triangulation of 20 random points with a random seed of 0

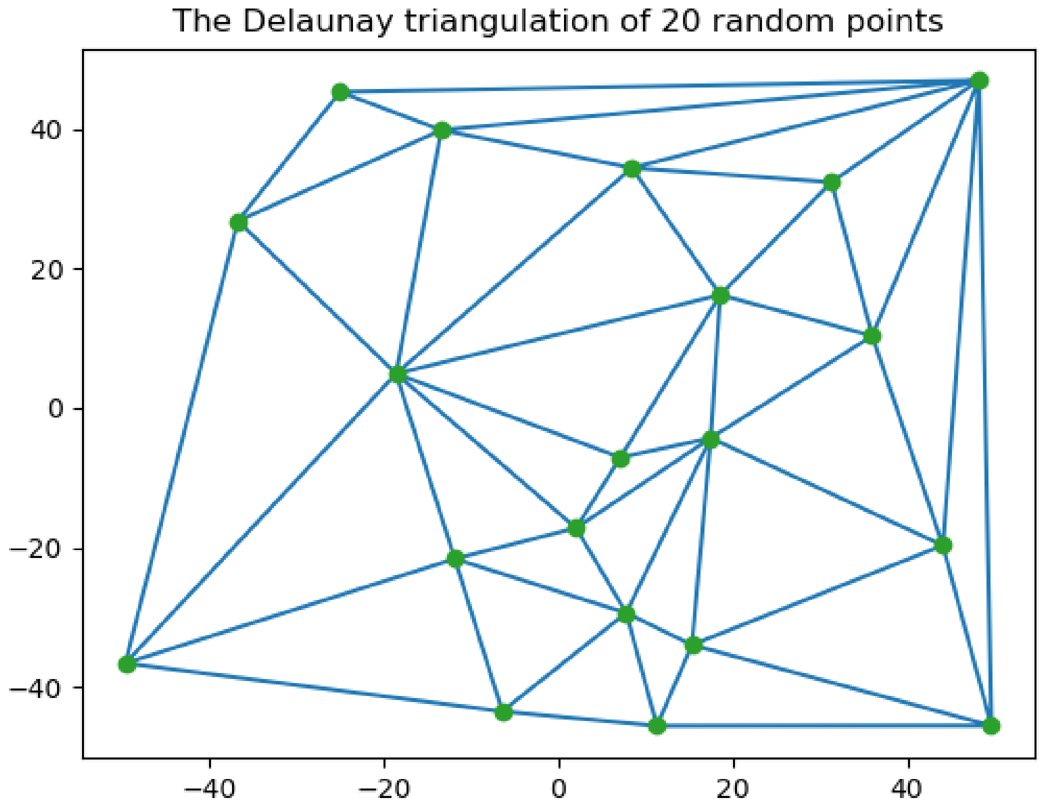

Figure 9: The Delaunay triangulation of 20 random points with a random seed of 10

```
$ python -m pip install virtualenv
$ python -m virtualenv .env
$ . .env/Scripts/activate
$ pip install requirements.txt
$ python delaunay.py --help
Usage: delaunay.py [OPTIONS]

  Compute the Delaunay triangulation of different sets of vertices

Options:
  -s, --seed INTEGER               the seed of the random number generator
  -n, --number INTEGER             the number of random points to be generated
                                   [default: 10]

  -x, --x-axis <FLOAT FLOAT>...    the minimum and maximum horizontal coordinate
                                   value  [default: -50.0, 50.0]

  -y, --y-axis <FLOAT FLOAT>...    the minimum and maximum vertical coordinate
                                   value  [default: -50.0, 50.0]

  -f, --filename FILENAME          optionally save the figure in PNG format
  --help                           Show this message and exit.
```

Figure 10: How to run the code

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

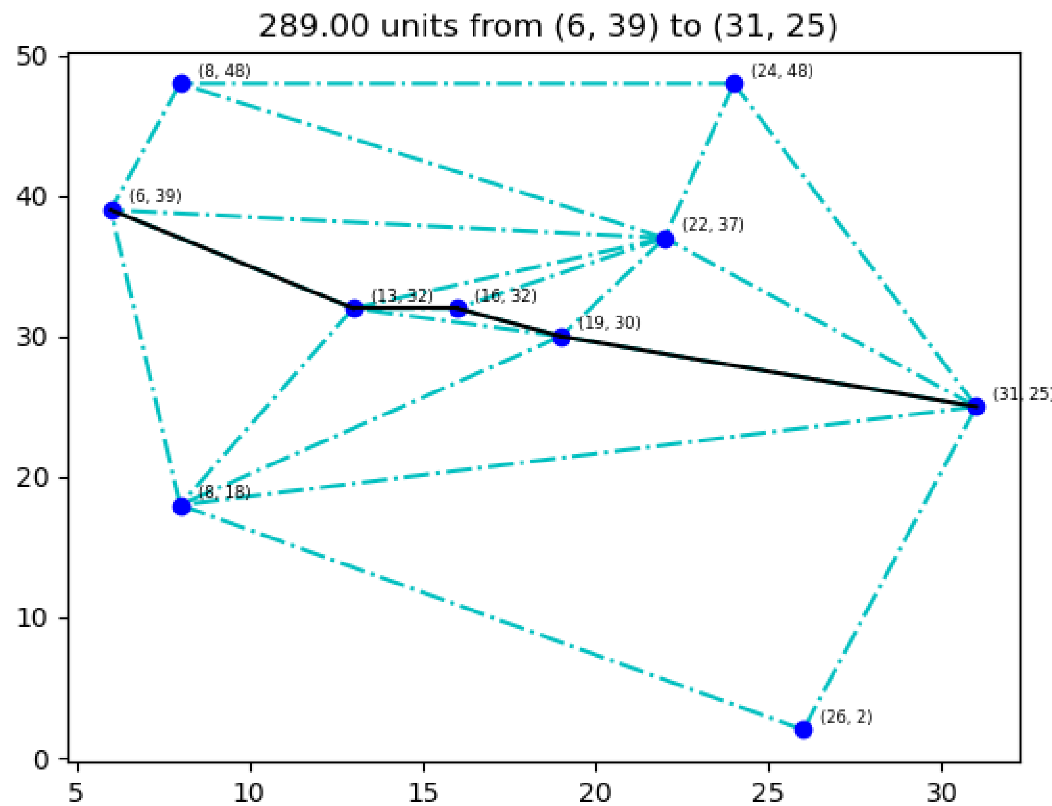

Figure 11: One of the shortest paths in a triangulation of 10 random points with a random seed of 0

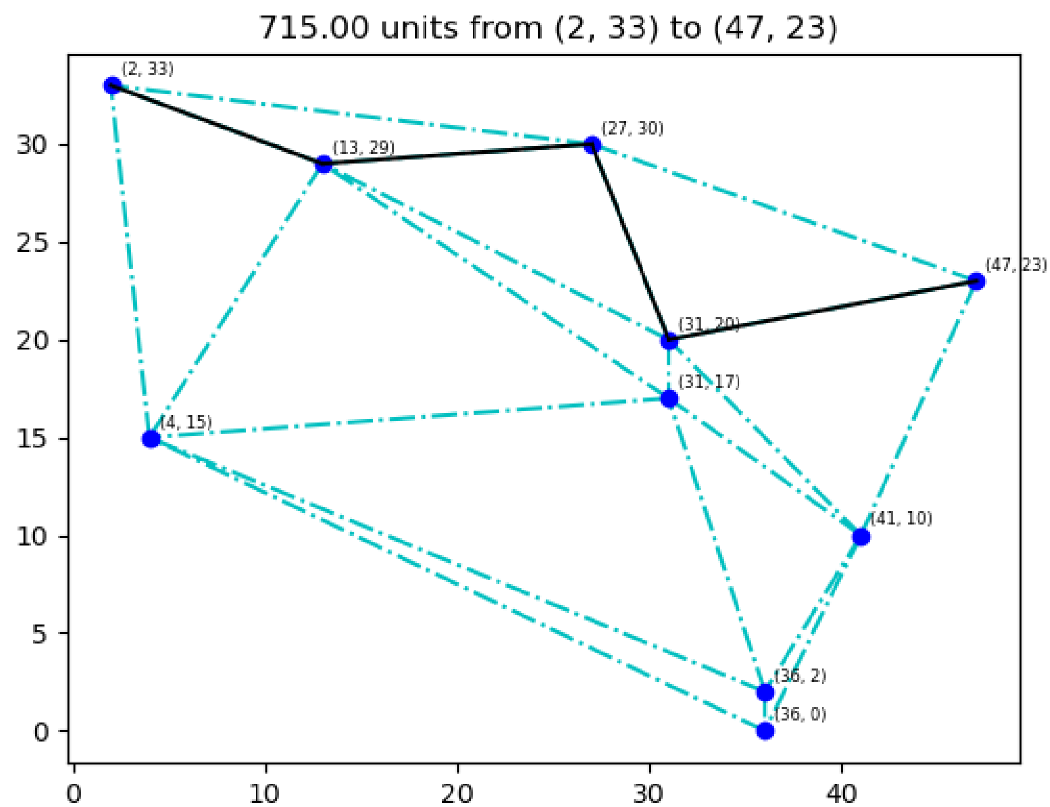

Figure 12: One of the shortest paths in a triangulation of 10 random points with a random seed of 10

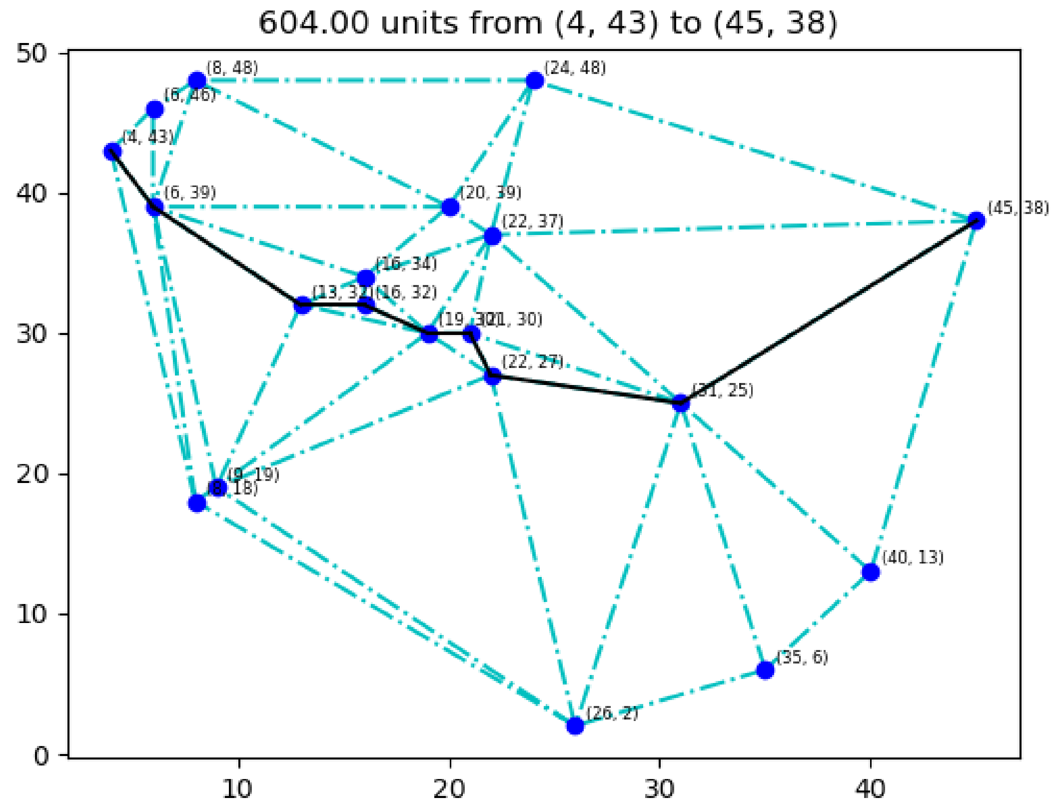

Figure 13: One of the shortest paths in a triangulation of 20 random points with a random seed of 0

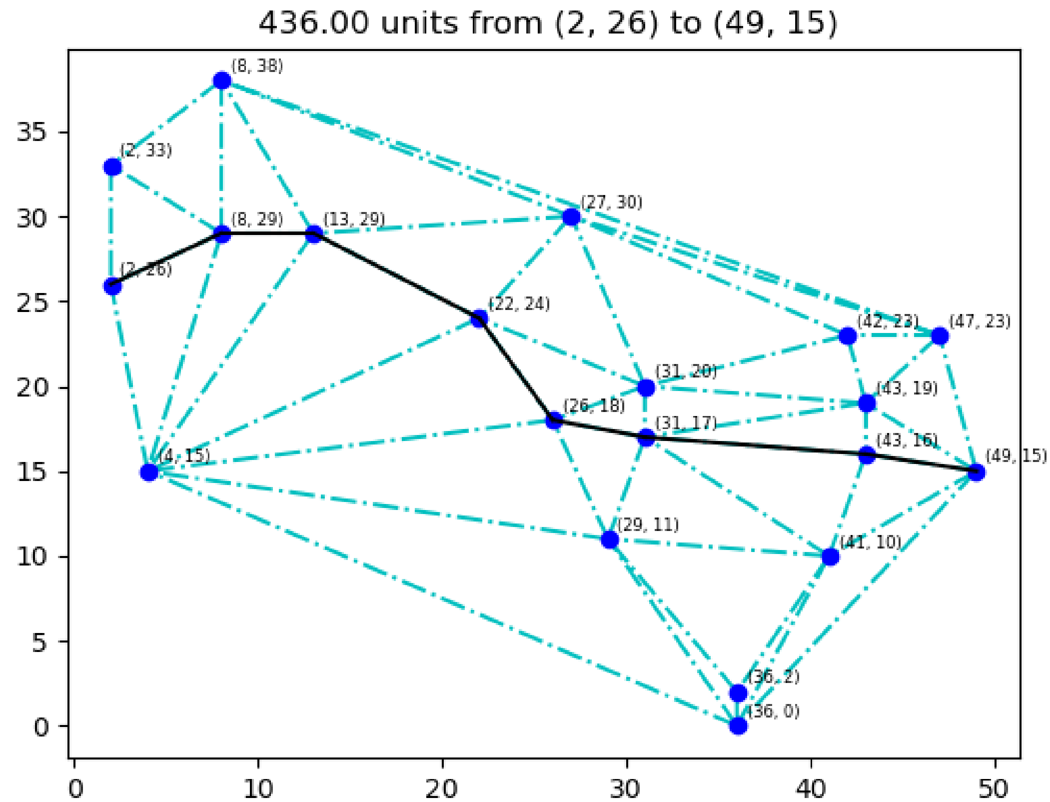

Figure 14: One of the shortest paths in a triangulation of 20 random points with a random seed of 10

```
$ python -m pip install virtualenv
$ python -m virtualenv .env
$ . .env/Scripts/activate
$ pip install requirements.txt
$ python shortest_path.py --help
Usage: shortest_path.py [OPTIONS]

  Compute the shortest path of different set of vertices in a tri-angulation

Options:
  -s, --seed INTEGER                  the seed of the random number generator
  -n, --number INTEGER                the number of random points to be generated
                                      [default: 10]

  -x, --x-axis INTEGER                the maximum horizontal coordinate value
                                      [default: 50]

  -y, --y-axis INTEGER                the maximum vertical coordinate value
                                      [default: 50]

  -m, --metric euclidean|manhattan
                                      the metric to be used when calculating the
                                      distance of two vertices  [default:
                                      euclidean]

  -b, --begin INTEGER                 the starting vertex  [default: 0]
  -e, --end INTEGER                   the ending vertex  [default: -1]
  -f, --filename FILENAME             optionally save the figure in PNG format
  --help                              Show this message and exit.
```

Figure 15: How to run the code

4. Experiment yourself with the *.encloses_point* and *.encloses* methods of the sympy.geometry module using polygons or circles to check if they contain certain points of your choice. Do the same with *contains_point* or *contains* points from the Path class from the libraries of matplotlib.path.

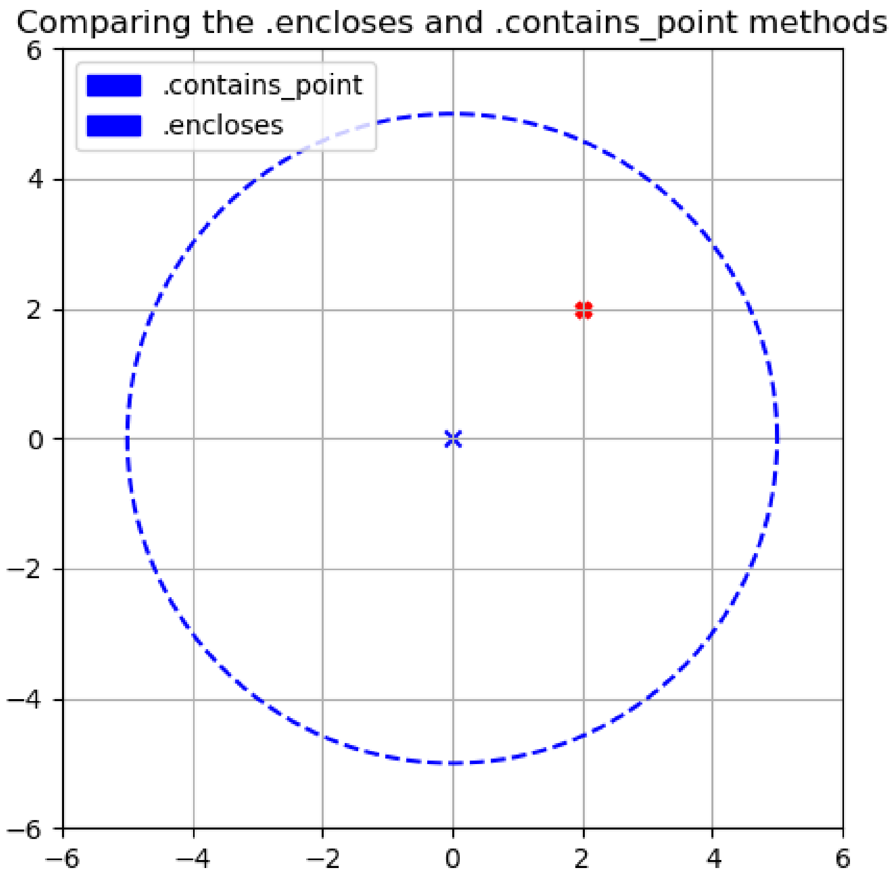

Figure 16: Using the *encloses* method

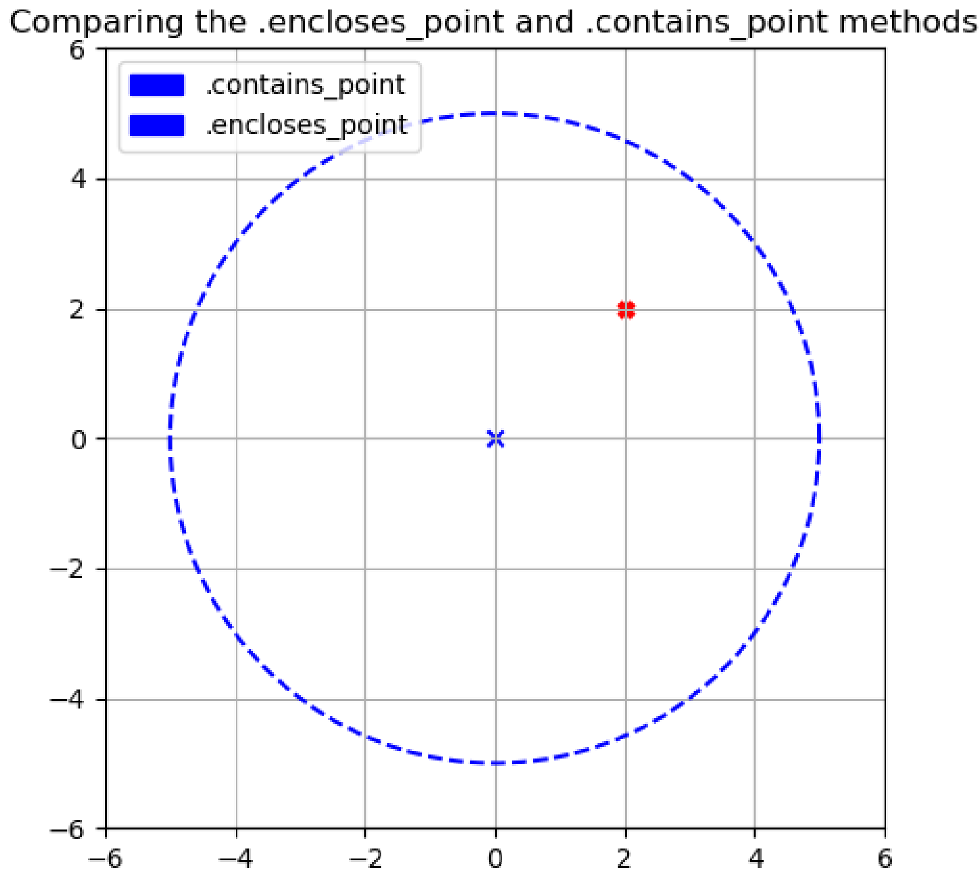

Figure 17: Using the *encloses_point* method

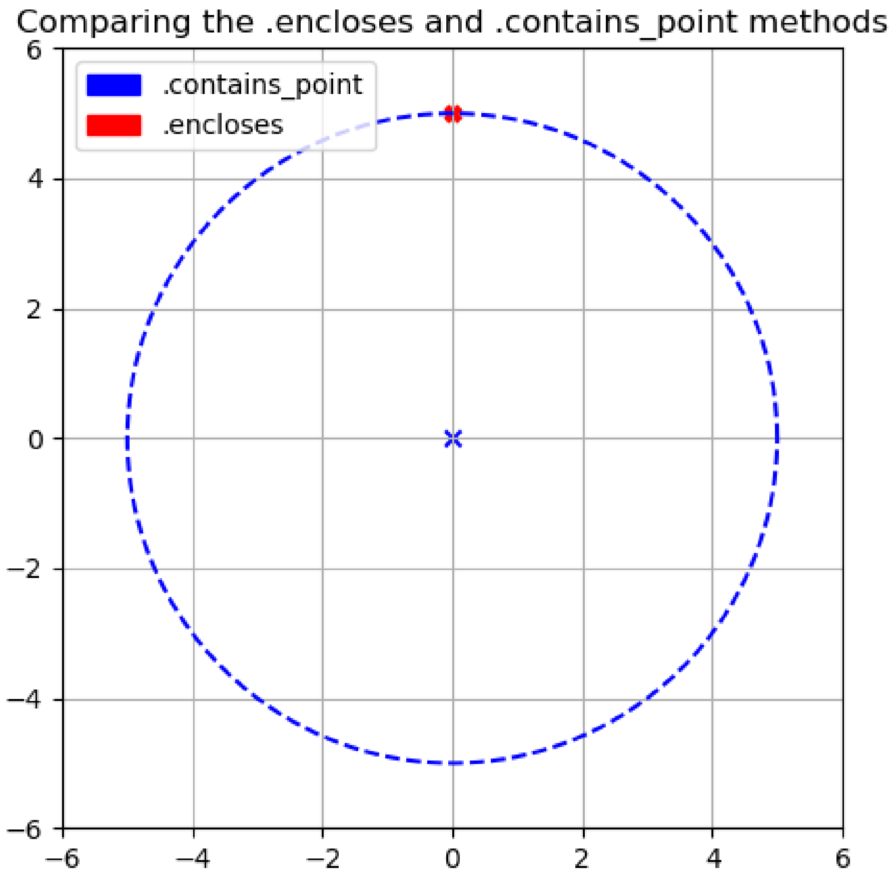

Figure 18: Comprtin the *encloses* and *contains* methods

```
$ python -m pip install virtualenv
$ python -m virtualenv .env
$ . .env/Scripts/activate
$ pip install requirements.txt
$ python encloses_contains.py --help
Usage: encloses_contains.py [OPTIONS]

  Comparing the `sympy.geometry` and `matplotlib.path` libraries

Options:
  -p, --point <FLOAT FLOAT>...    the target point  [default: 2, 2]
  -c, --circle <FLOAT FLOAT FLOAT>...
                                  the circle's center and radius  [default: 0,
                                  0, 5]

  -s, --sympy [encloses_point|encloses]
                                  the entity method  [default: encloses]
  -f, --filename FILENAME         optionally save the figure in PNG format
  --help                          Show this message and exit.
```

Figure 19: How to run the code

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

Δεδομένου ενός συνόλου σημείων ορίζεται το αντίστοιχο Voronoi διάγραμμα.

Θα επιχειρήσουμε να υπολογίσουμε τα Voronoi διαγράμματα διάφορων συνόλων σημείων στον δισδιάστατο χώρο, με τη βοήθεια του αλγορίθμου K κοντινότερων γειτόνων.

Δεδομένου ενός συνόλου εκπαίδευσης P, του οποίου το Voronoi διάγραμμα επιθυμούμε να υπολογίσουμε, ορίζουμε το meshgrid το οποίο ορίζεται από τα σημεία

$$(\min_{\forall p \in P} p.x), \min_{\forall p \in P} p.y))$$

$$(\max_{\forall p \in P} p.x), \max_{\forall p \in P} p.y))$$

Αφού εκπαιδεύσουμε το μοντέλο μας, κατηγοριοποιούμε κάθε σημείο που ανήκει στο meshgrid, έτσι ώστε να προκύψει το Voronoi διάγραμμα.

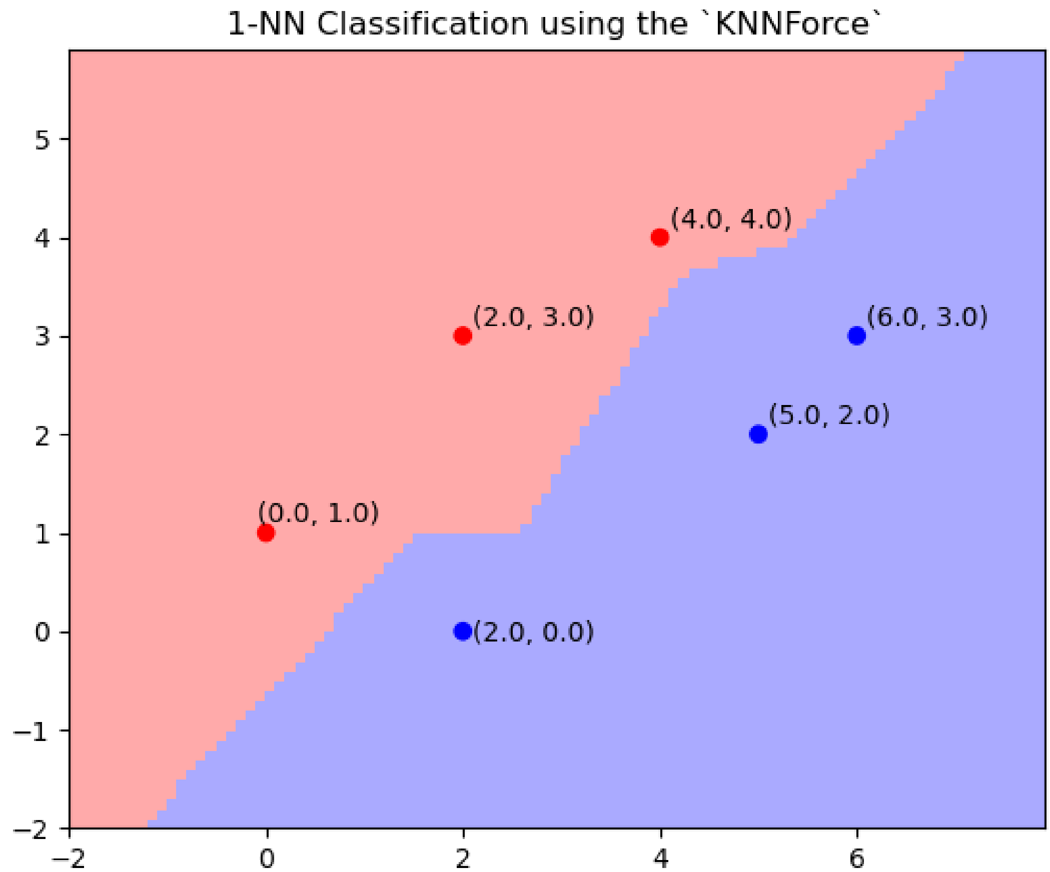

Figure 20: Using brute force in order to find the single closest neighbor of 80 points

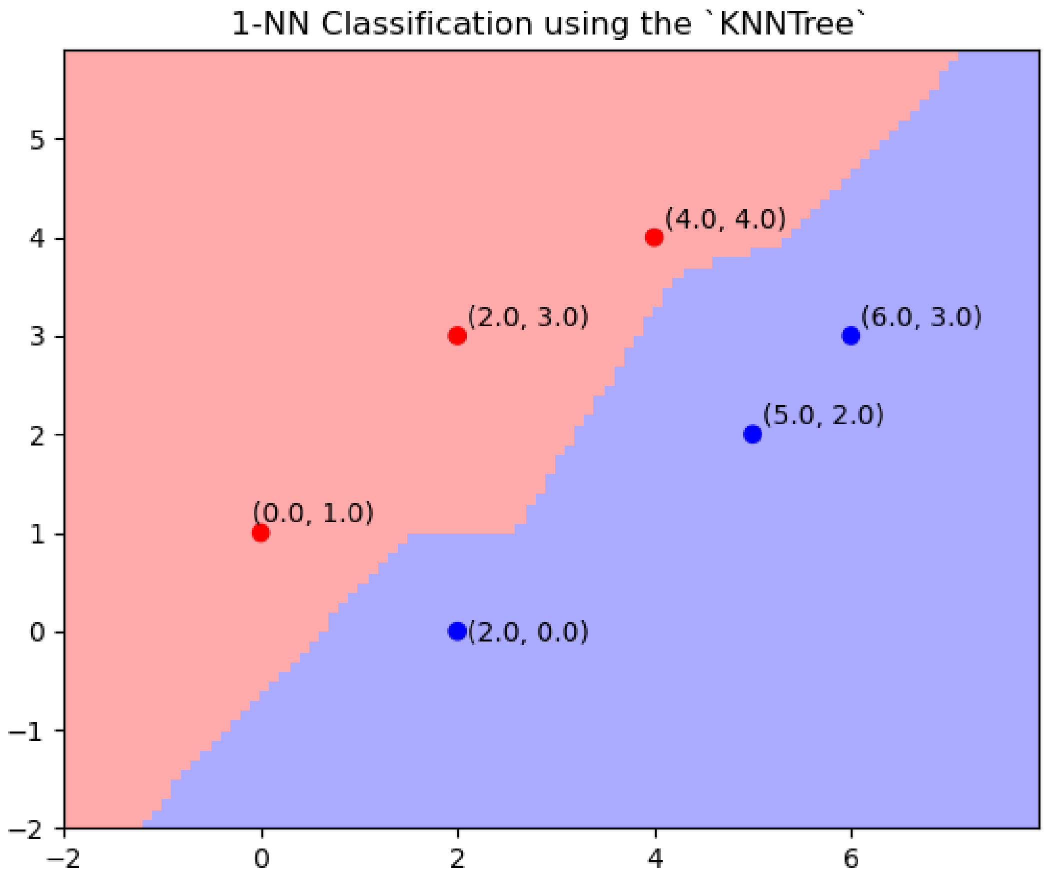

Figure 21: Using KD-Tree neighborhood look-up in order to find the single closest neighbor of 80 points

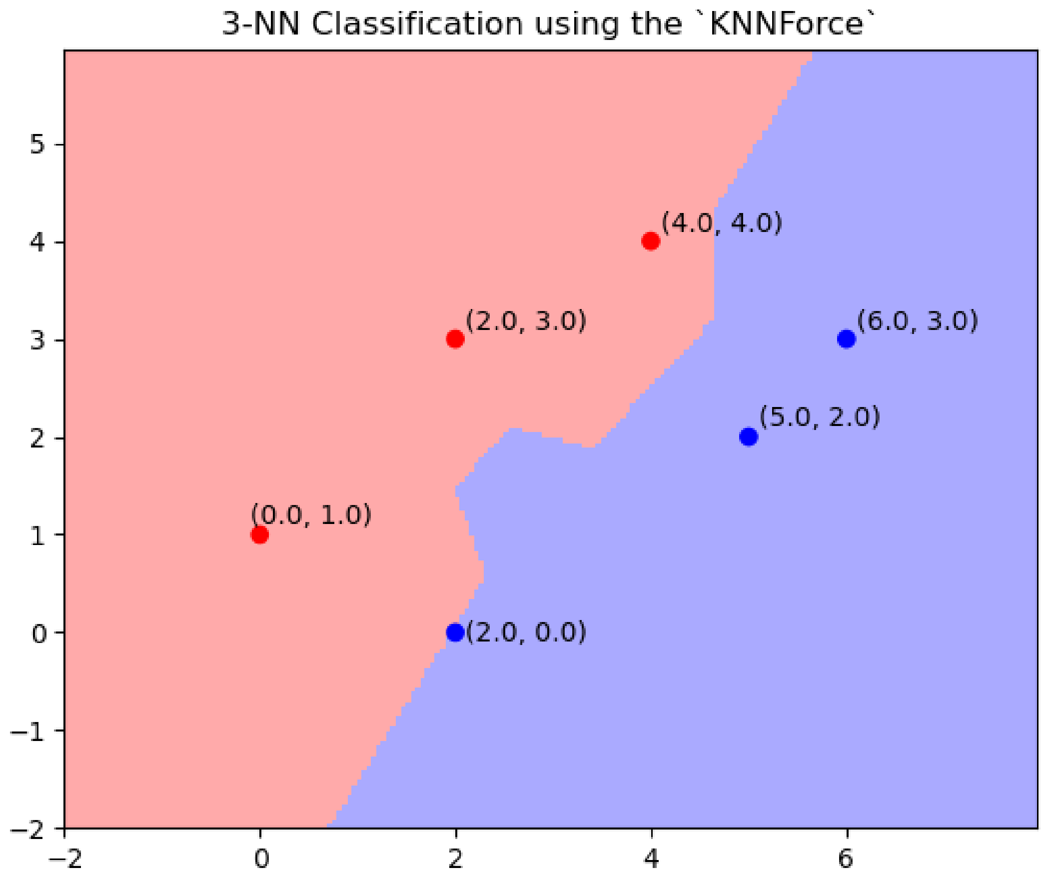

Figure 22: Using brute force in order to find the 3 closest neighbors of 160 points

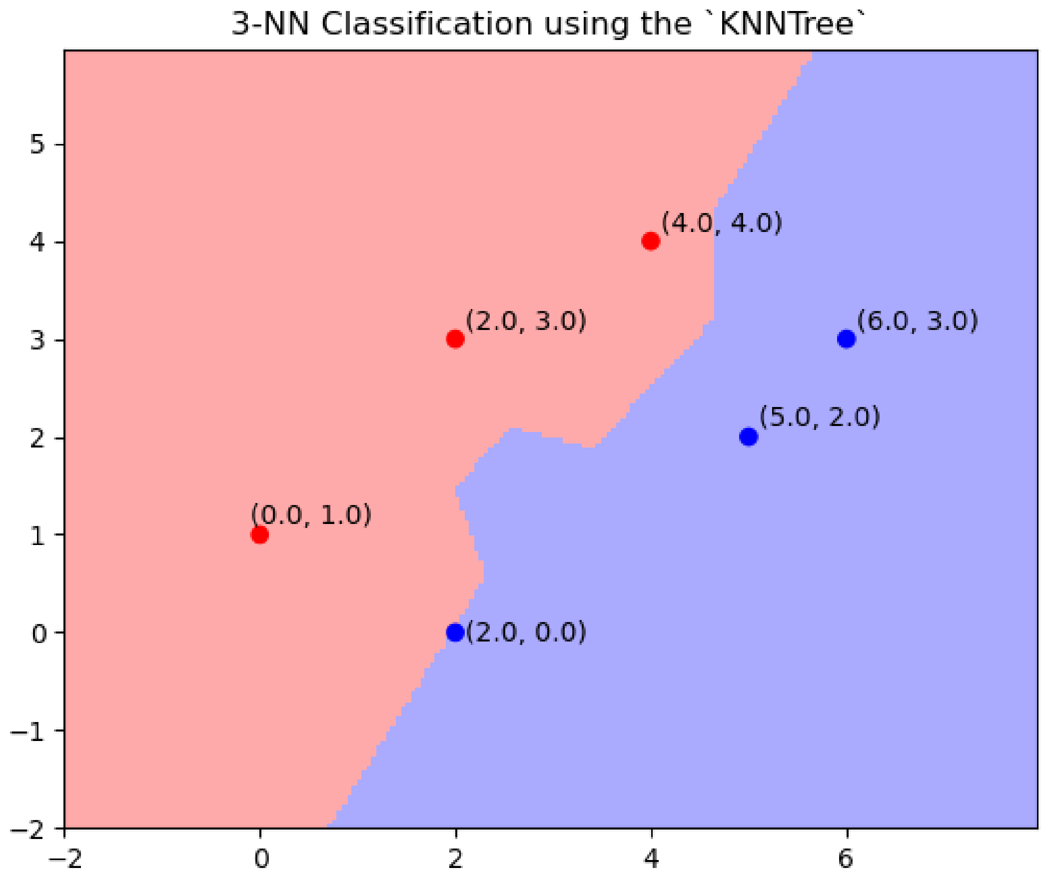

Figure 23: Using KD-Tree neighborhood look-up in order to find the 3 closest neighbors of 160 points

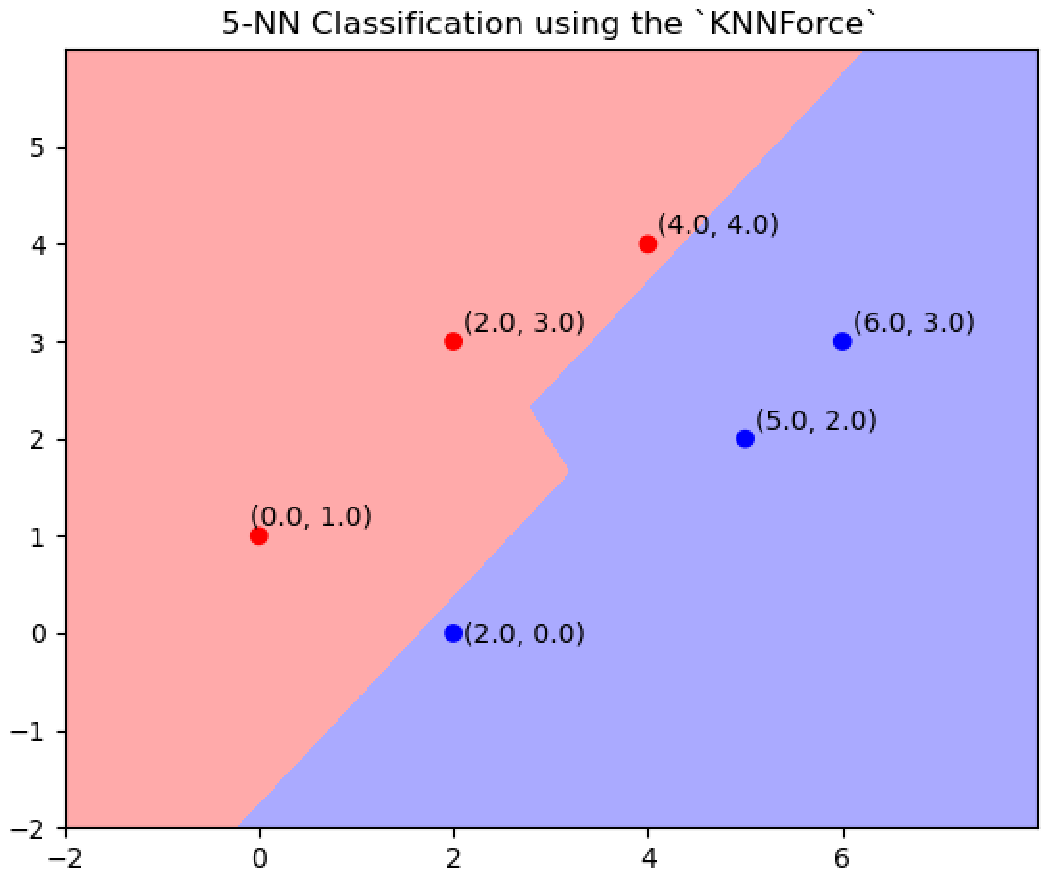

Figure 24: Using brute force in order to find the 5 closest neighbors of 1600 points

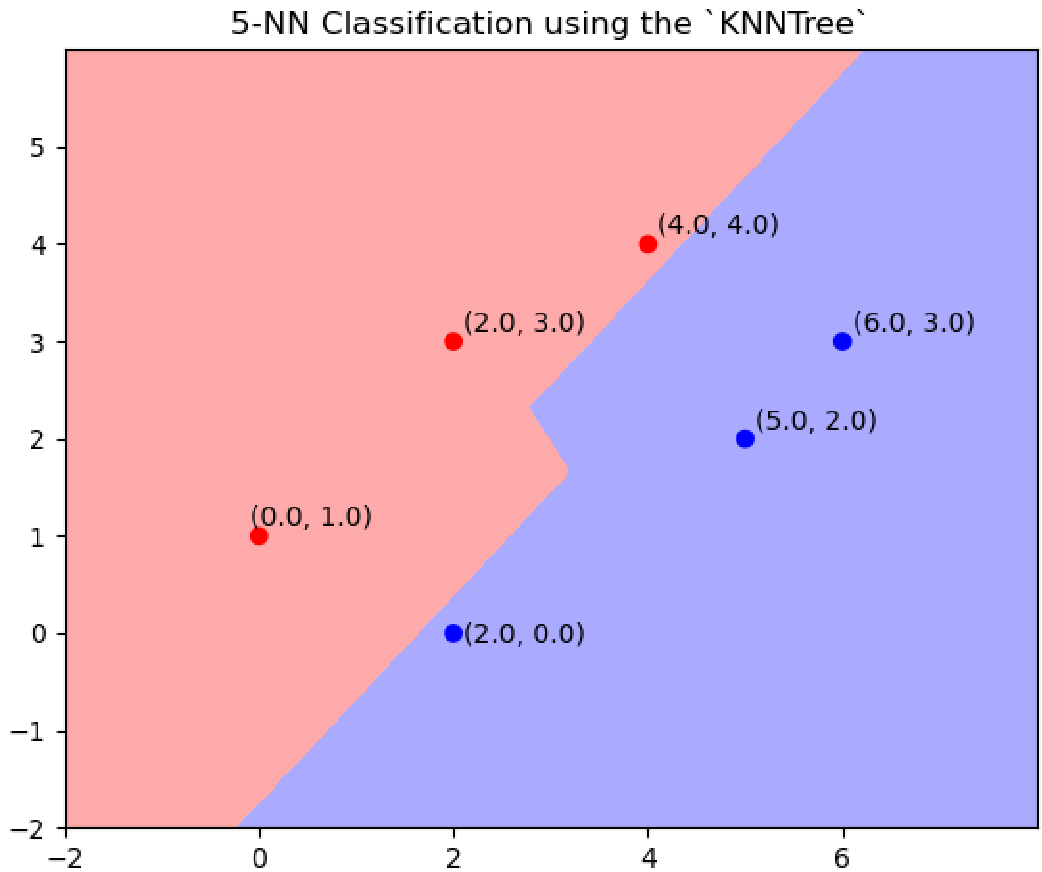

Figure 25: Using KD-Tree neighborhood look-up in order to find the 5 closest neighbors of 1600 points

Παρ' ότι η μέθοδος των KD-Δέντρων είναι προσεγγιστική και δεν εγγυάται ότι θα βρεί τους K πραγματικά κοντινότερους γείτονες, στην προκείμενη περίπτωση η απόκλιση του τελικού αποτελέσματος σε σχέση με τη μέθοδο ωμής βίας είναι ουσιαστικά ανύπαρκτη.

Ωστόσο, αυτό που είναι σίγουρο είναι πώς η μέθοδος των KD-Δέντρων είναι σημαντικά γρηγορότερη, όπως φαίνεται και από τα παρακάτω δεδομένα.

Type	Neighbors	Predictions	Time	Efficiency
force	1	800	43271.45770	1
force	1	160	2870.635500	1
force	1	80	477.2699000	1
force	3	800	54532.69620	1
force	3	160	2157.242100	1
force	3	80	546.4535000	1
force	5	1600	252786.1562	1
force	5	800	63132.58790	1
tree	1	800	20512.63240	2.1095029080714185
tree	1	160	899.3549000	3.1918828707109950
tree	1	80	225.0998000	2.1202591028512687
tree	3	800	21523.35190	2.5336525859617620
tree	3	160	1015.666200	2.1239675988036226
tree	3	80	211.0624000	2.5890613392058460
tree	5	1600	103242.2123	2.4484767477226947
tree	5	800	21571.52390	2.9266633267388213

```
$ python -m pip install virtualenv
$ python -m virtualenv .env
$ . .env/Scripts/activate
$ pip install requirements.txt
$ python kdtree.py --help
Usage: kdtree.py [OPTIONS]

  Perform various KD-Tree associated experiments

Options:
  -n, --neighbors INTEGER       the number of neighbors  [default: 1]
  -t, --train <FLOAT FLOAT>...  the training data  [default: (0, 1), (2, 3),
                                (4, 4), (2, 0), (5, 2), (6, 3)]

  -l, --labels INTEGER          a list of labels, one for each training
                                instance  [default: 0, 0, 0, 1, 1, 1]

  -m, --mode force|tree         brute force or kd-tree based k-neighborhood
                                lookup  [default: force]

  -s, --meshgrid-step FLOAT     the meshgrid step determines the number of
                                points, whose labels are going to be predicted
                                by KNN, so that the underlying Voronoi diagram
                                can be illustrated  [default: 0.02]

  -f, --filename FILENAME       optionally save the figure in PNG format
  --help                        Show this message and exit.
```

Figure 26: How to run the code