Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

MCS 507 Lecture 31
Mathematical, Statistical and Scientific Software
Jan Verschelde, 1 November 2023

Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

computational geometry

Computational geometry studies algorithms for geometric problems.

We distinguish three classes of problems:

- **Static :** compute an object from given inputs.
  Examples: convex hulls, triangulations, Voronoi diagrams.
- **Query :** given a search space and a query,
  find objects in the search space that satisfy the query.
- **Dynamic :** similar problems as static and query,
  but the input changes and objects are inserted and deleted.

Applications : finite element methods, robot motion planning, etc.

computational geometry software

We follow *Mastering SciPy* by Francisco J. Blanco-Silva, Packt Publishing 2015.

We introduce three software packages:

- The `spatial` package of `scipy` for spatial algorithms and data structures contains KDTree and query methods.
- Mayavi: 3D scientific data visualization and plotting, provides `mlab`, for 3D plotting in Python.
- Triangle: a mesh generator by Jonathan Richard Shewchuck, used through the wrapper written by Dzhelil Rufat, available at `http://dzhelil.info/triangle/index.html`.

The Computational Geometry Algorithms Library

http://www.cgal.org

CGAL is a software project that provides easy access and reliable geometric algorithms in the form of a C++ library. CGAL provides an extensive collection of examples.

Its development is funded by academia and companies, dual licensing:

- open source: Lesser GPL (LGPL) and GPL;
- a commercial license from the GeometryFactory.

Installation:

- CGAL is available via package managers.
- There is a Python interface, install with pip or conda.

CGAL Made More Accessible, by Nir Goren, Efi Fogel, and Dan Halperin, `arxiv:2202.13889v2 [cs.CG] 7 Jun 2023`.

Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

the k-d tree problem

Problem statement:

> Input: a set S of points.
> Output: a k-d tree to store the points in S.

A k-d tree is a recursive data structure:

1. The points are split on one coordinate.
2. The two halves are splitted on the next coordinate.

The k-d tree allows for fast point location.

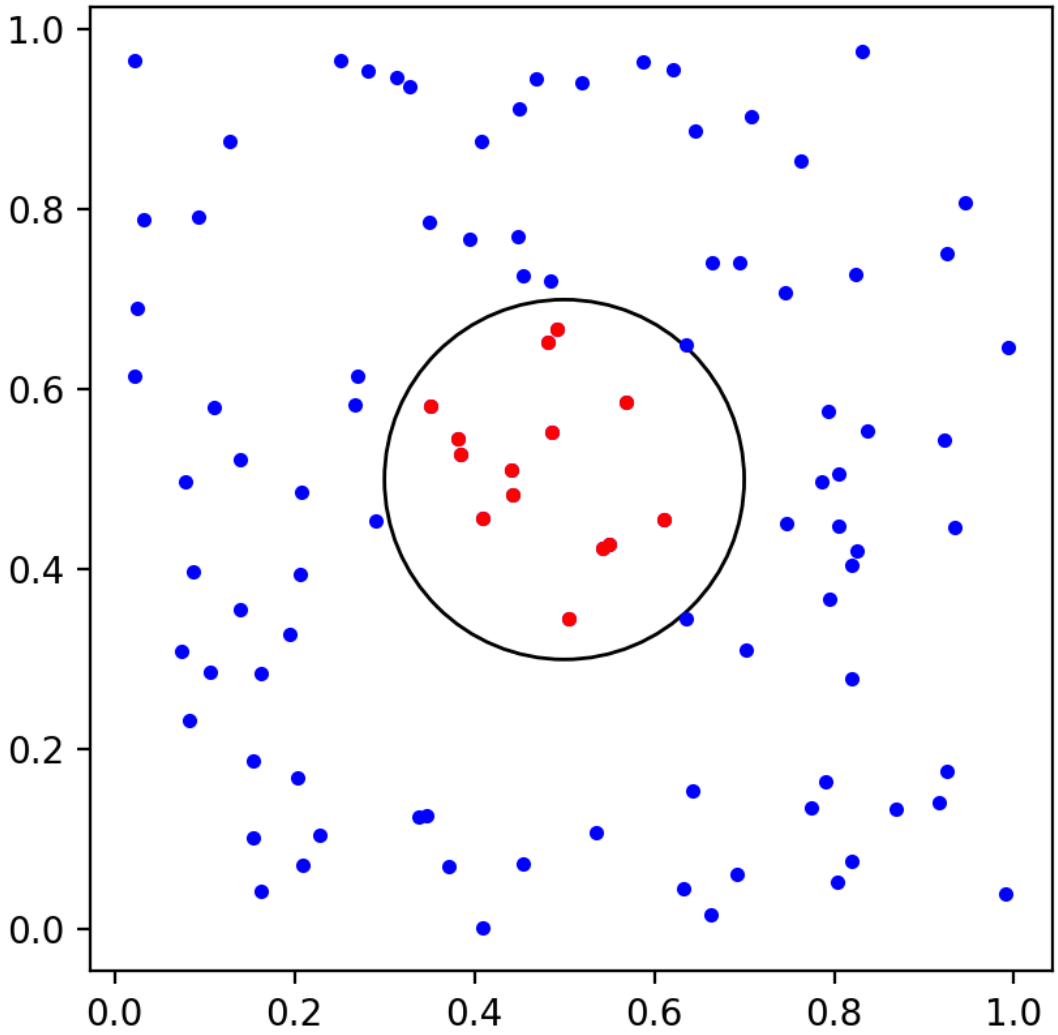
a k-d tree of 100 random points in the plane

The query region is a disk centered at (0.5, 0.5), with radius 0.2.

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

points = np.random.rand(100, 2)
tree = KDTree(points)
disk = tree.query_ball_point((0.5, 0.5), r=0.2, p=2.0)
print(disk)
fig = plt.figure()
plt.axes().set_aspect('equal')
plt.plot(points[:,0], points[:,1], 'b.')
for idx in disk:
    x, y = points[idx]
    plt.plot(x, y, 'r.')
circle = plt.Circle((0.5, 0.5), 0.2, fill=False)
plt.gca().add_artist(circle)
plt.show()
```

the plot of the points in the query disk



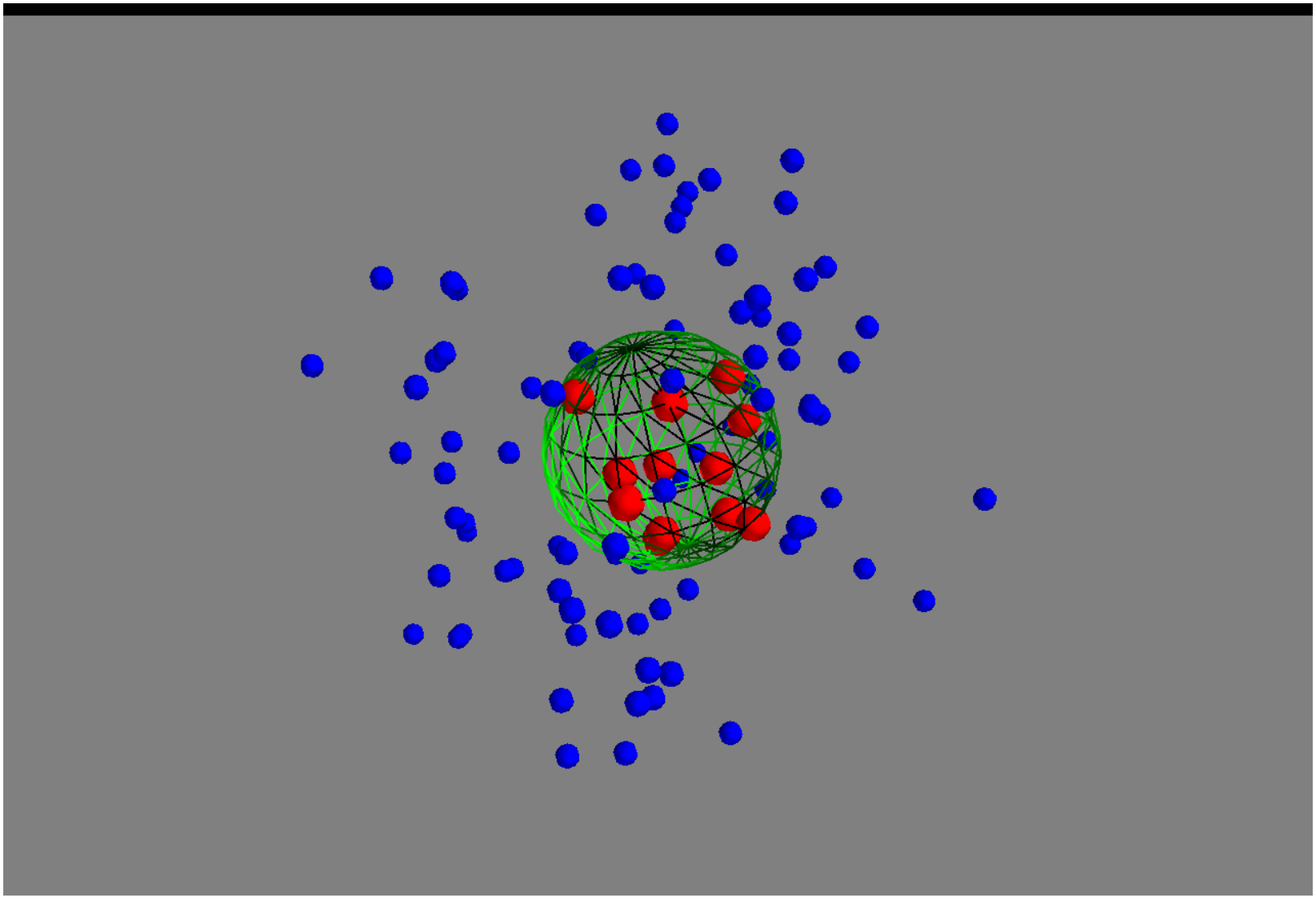
a k-d tree of 100 random points in 3-space

The query region is a ball centered at (0.5, 0.5, 0.5), with radius 0.25.

```
import numpy as np
from scipy.spatial import KDTree
from mayavi import mlab

points = np.random.rand(100, 3)
tree = KDTree(points)
ball = tree.query_ball_point( (0.5, 0.5, 0.5), r=0.25, p=2.0)
print(ball)
mlab.points3d(points[:,0], points[:,1], points[:,2],
    color=(0,0,1), scale_factor=0.05)
for idx in ball:
    x, y, z = points[idx]
    mlab.points3d(x, y, z, color=(1,0,0), scale_factor=0.075)
u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]
x = 0.5+0.25*np.cos(u)*np.sin(v)
y = 0.5+0.25*np.sin(u)*np.sin(v)
z = 0.5+0.25*np.cos(v)
mlab.mesh(x, y, z, transparent=True, color=(0,1,0),
    representation='wireframe')
mlab.show()
```

the plot of the points in the query ball



exploring the nodes in the k-d tree

```
import numpy as np
from scipy.spatial import KDTree

points = np.random.rand(100, 2)
tree = KDTree(points)
print('coordinate on split :', tree.tree.__dict__['split_dim'])
print('split value :', tree.tree.__dict__['split'])
print('number of children :', tree.tree.__dict__['children'])
left = tree.tree.__dict__['less']
print('-> left coordinate on split :', left.__dict__['split_dim'])
print('   split value :', left.__dict__['split'])
print('   number of children :', left.__dict__['children'])
left1 = left.__dict__['less']
print('   -> left coordinate on split :', left1.__dict__['split_dim'])
print('      split value :', left1.__dict__['split'])
print('      number of children :', left1.__dict__['children'])
right1 = left.__dict__['greater']
print('   -> right coordinate on split :', right1.__dict__['split_dim'])
print('      split value :', right1.__dict__['split'])
print('      number of children :', right1.__dict__['children'])
right = tree.tree.__dict__['greater']
print('-> right coordinate on split :', right.__dict__['split_dim'])
print('   split value :', right.__dict__['split'])
print('   number of children :', right.__dict__['children'])
left2 = right.__dict__['less']
print('   -> left coordinate on split :', left2.__dict__['split_dim'])
print('      split value :', left2.__dict__['split'])
print('      number of children :', left2.__dict__['children'])
right2 = right.__dict__['greater']
print('   -> right coordinate on split :', right2.__dict__['split_dim'])
print('      split value :', right2.__dict__['split'])
print('      number of children :', right2.__dict__['children'])
```

output of the script

```
coordinate on split : 0
split value : 0.5002283563911739
number of children : 100
-> left coordinate on split : 1
   split value : 0.5029027989408748
   number of children : 47
   -> left coordinate on split : 0
      split value : 0.2518749666965 4996
      number of children : 27
   -> right coordinate on split : 0
      split value : 0.2518749666965 4996
      number of children : 20
-> right coordinate on split : 1
   split value : 0.5029027989408748
   number of children : 53
   -> left coordinate on split : 0
      split value : 0.7485817460857979
      number of children : 24
   -> right coordinate on split : 0
      split value : 0.7485817460857979
      number of children : 29
```

Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

searching in a Delaunay triangulation

A Delaunay triangulation T of a point set S
maximizes the minimal angle over all triangulations of S.

Query a Delaunay triangulation with a point:

> Input: a Delaunay triangulation T and a point $\mathbf{p}$.
> Output: the triangle $\Delta \in T$ for which $\mathbf{p} \in \Delta$.

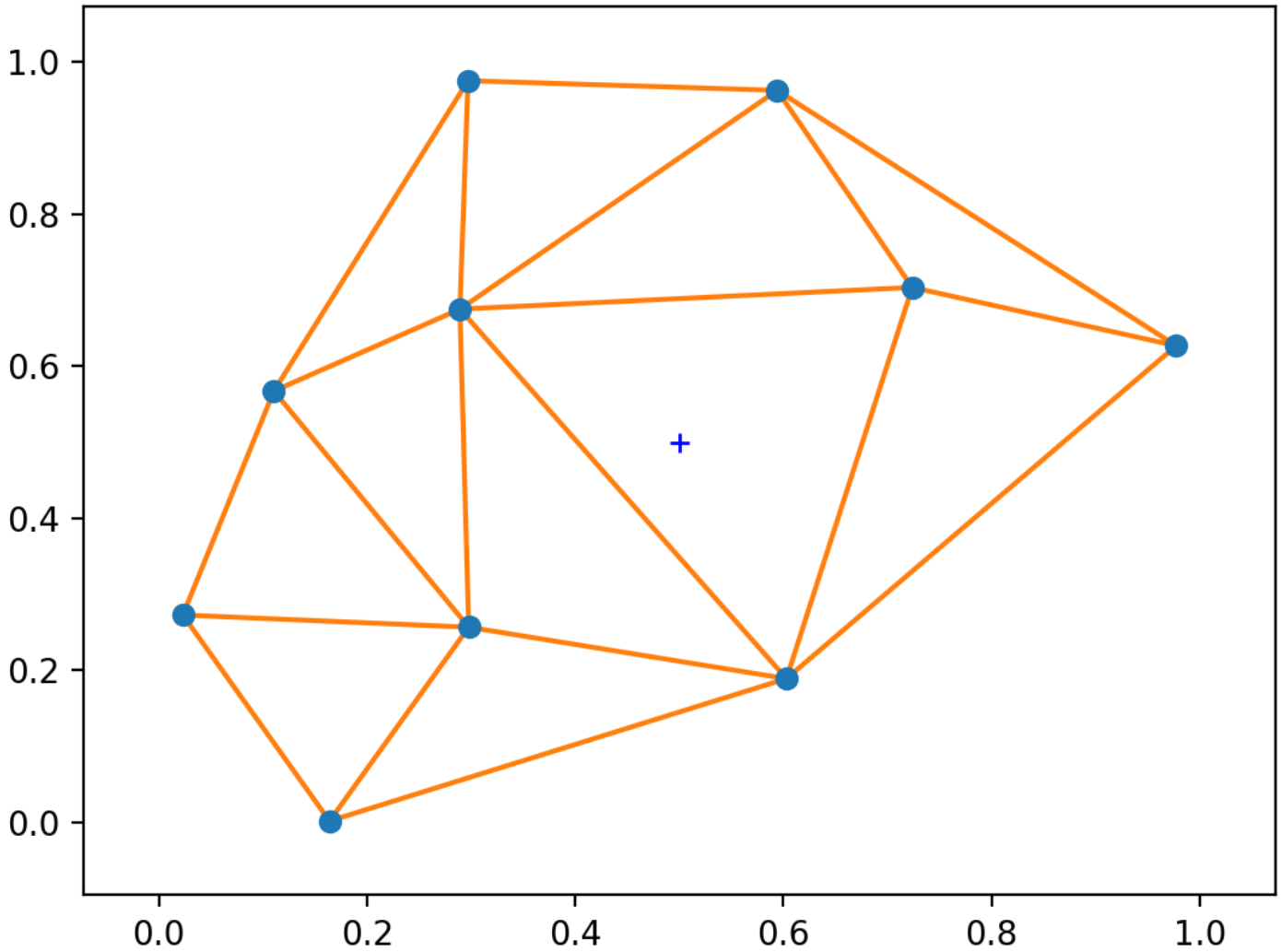
We do this query with `tsearch` of `scipy.spatial`.

a script to search a Delaunay triangulation

```
import numpy as np
from scipy.spatial import Delaunay, delaunay_plot_2d
from scipy.spatial import tsearch
from matplotlib import pyplot as plt

points = np.random.rand(10, 2)
tri = Delaunay(points)
point = [0.5, 0.5]
idx = tri.find_simplex(point)
result = tri.simplices[idx]
print('Searched for the point (0.5, 0.5) ...')
print('Found in simplex with index :', idx)
print('spanned by')
for pt in result:
    print('point', pt, ':', points[pt])
fig = plt.figure()
axs = fig.add_subplot()
delaunay_plot_2d(tri, ax=axs)
axs.plot(point[0], point[1], 'b+')
plt.show()
```

a Delaunay triangulation and a point



output of the script

```
Searched for the point (0.5, 0.5) ...
Found in simplex with index : 4
spanned by
point 0 : [0.60290533 0.18915295]
point 2 : [0.72373128 0.703351  ]
point 5 : [0.28927568 0.6748418 ]
```

searching in a Voronoi diagram

A Voronoi Diagram V of a point set S is a partition of the plane in cells so that all points in the same cell are closest to the same point of S.

Query a Voronoi diagram with a point:

> Input: a Voronoi diagram V and a point $\mathbf{p}$.
> Output: the cell $C \in V$ for which $\mathbf{p} \in C$.

We do this query as follows.

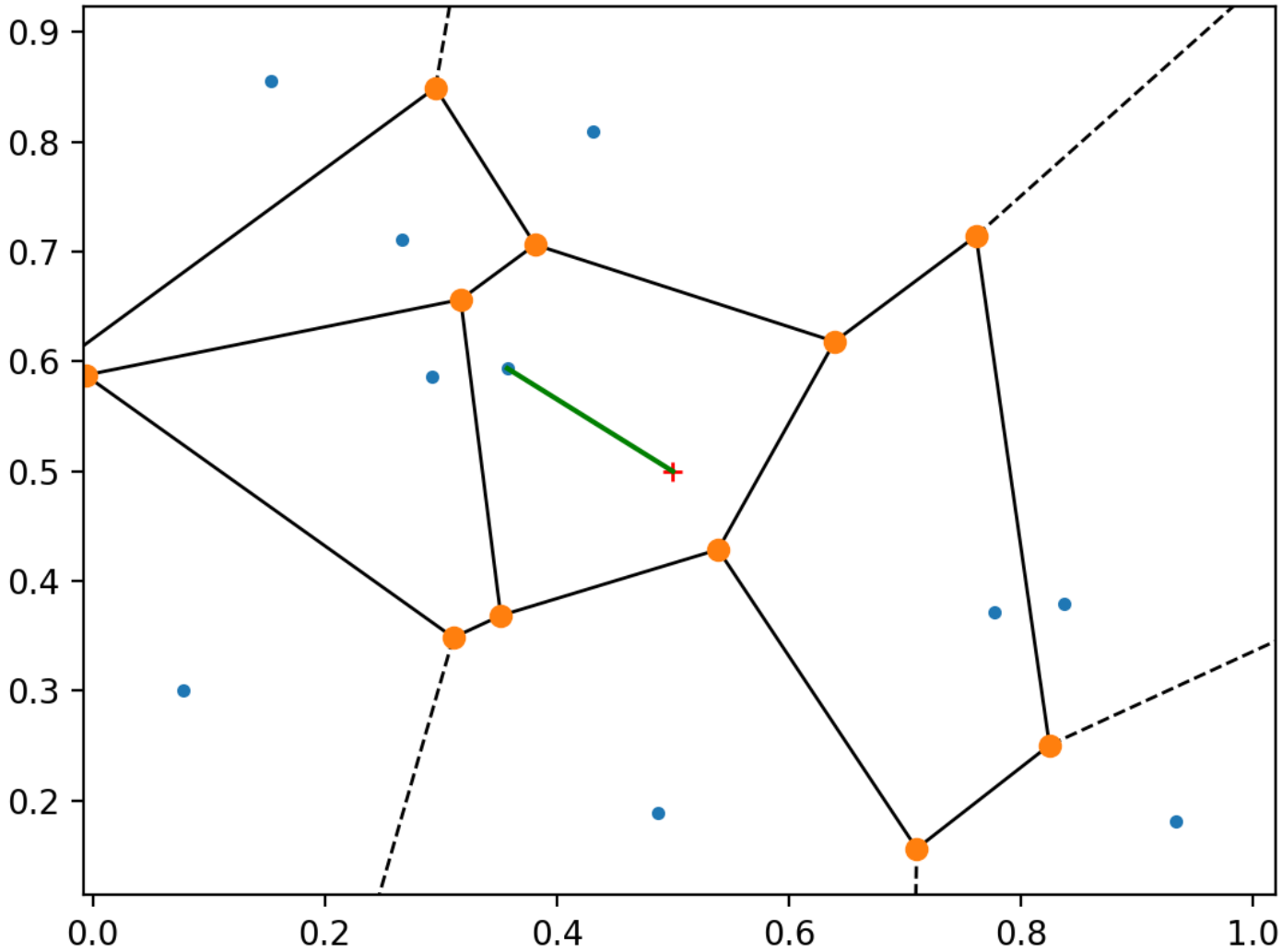
1. Make a k-d tree T of the sites S.
2. Applying the method `query` to T with $\mathbf{p}$ returns the site $\mathbf{s}$ closest to $\mathbf{p}$.
3. With $\mathbf{s}$ we identify the cell C in V, for which $\mathbf{p} \in C$.

a script to search a Voronoi diagram

```
import numpy as np
from scipy.spatial import Voronoi, voronoi_plot_2d
from scipy.spatial import KDTree
from matplotlib import pyplot as plt

points = np.random.rand(10, 2)
vor = Voronoi(points)
tree = KDTree(points)
point = [0.5, 0.5]
dst, idx = tree.query(point)
print('Searched for the point (0.5, 0.5) ...')
print('index of the nearest site :', idx)
print('distance of the nearest site :', dst)
print('coordinates of the nearest site :')
nearest = points[idx]
print(nearest)
fig = plt.figure()
axs = fig.add_subplot()
voronoi_plot_2d(vor, ax=axs)
axs.plot(point[0], point[1], 'r+')
axs.plot([point[0], nearest[0]], [point[1], nearest[1]], 'g-')
plt.show()
```

a Voronoi diagram and a point



output of the script

```
Searched for the point (0.5, 0.5) ...
index of the nearest site : 9
distance of the nearest site : 0.1705793713560641
coordinates of the nearest site :
[0.35747711 0.59372592]
```

Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

dynamic problems

In dynamic problems, the input changes.

To increment a Voronoi diagram with a point, we do:

```
vor = Voronoi(points, incremental=True)
vor.add_points([[x, y]])
```

To increment a Delaunay triangulation with a point, we do:

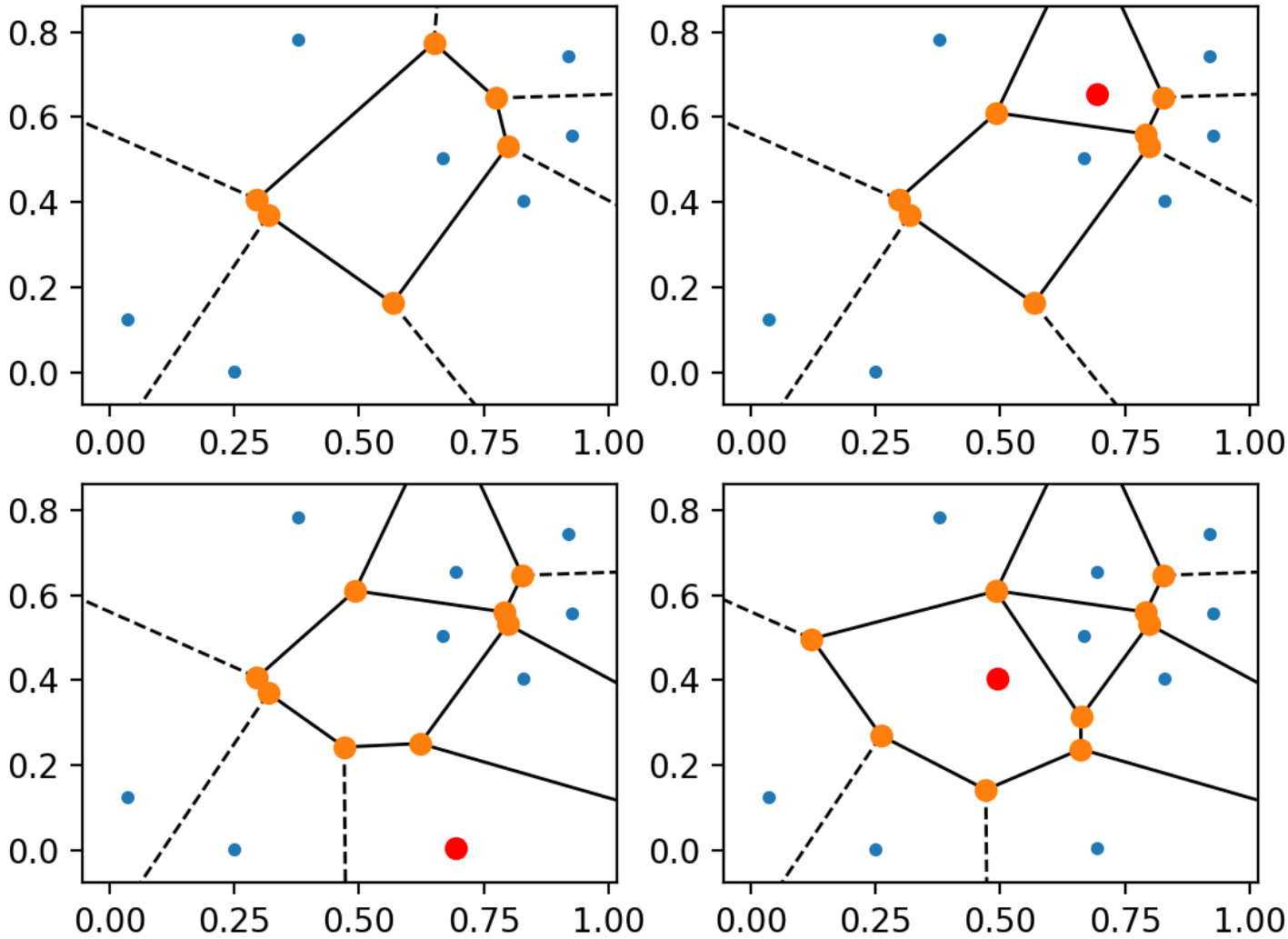
```
tri = Delaunay(points, incremental=True)
tri.add_points([[x, y]])
```

a script to increment a Voronoi diagram

```
import numpy as np
from scipy.spatial import Voronoi, voronoi_plot_2d
from matplotlib import pyplot as plt

points = np.random.rand(7, 2)
vor = Voronoi(points, incremental=True)
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax1.set_xlim(0.0, 1.1)
ax1.set_ylim(0.0, 1.1)
voronoi_plot_2d(vor, ax=ax1)
newpoints = np.random.rand(3, 2)
for k in range(3):
    axk = fig.add_subplot(2, 2, k+2)
    axk.set_xlim(0.0, 1.1)
    axk.set_ylim(0.0, 1.1)
    x, y = newpoints[k]
    vor.add_points([[x, y]])
    voronoi_plot_2d(vor, ax=axk)
    axk.plot(x, y, 'ro')
plt.show()
```

incrementing a Voronoi diagram

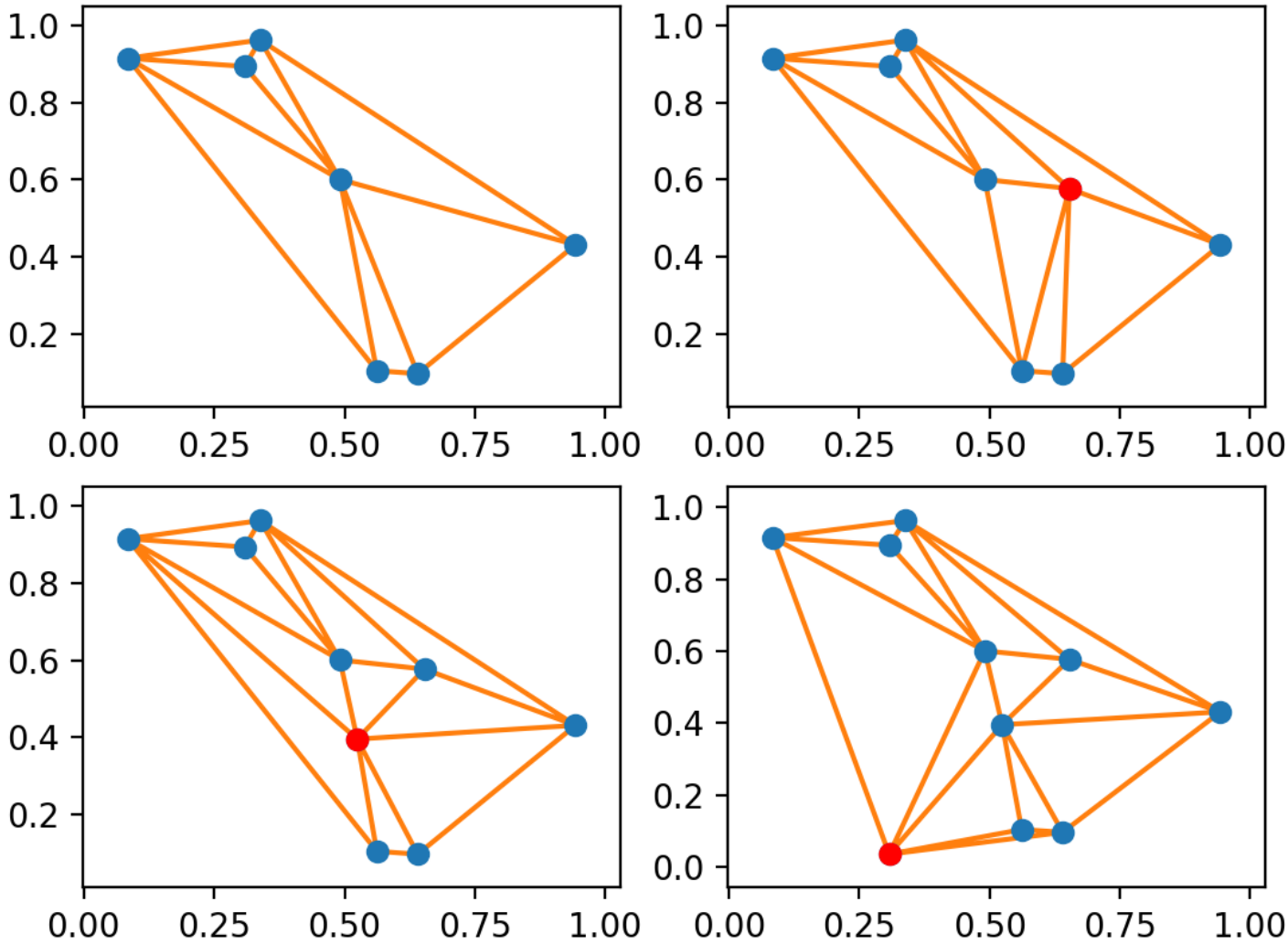


a script to increment a Delaunay triangulation

```
import numpy as np
from scipy.spatial import Delaunay, delaunay_plot_2d
from matplotlib import pyplot as plt

points = np.random.rand(7, 2)
tri = Delaunay(points, incremental=True)
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax1.set_xlim(0.0, 1.1)
ax1.set_ylim(0.0, 1.1)
delaunay_plot_2d(tri, ax=ax1)
newpoints = np.random.rand(3, 2)
for k in range(3):
    axk = fig.add_subplot(2, 2, k+2)
    axk.set_xlim(0.0, 1.1)
    axk.set_ylim(0.0, 1.1)
    x, y = newpoints[k]
    tri.add_points([[x, y]])
    delaunay_plot_2d(tri, ax=axk)
    axk.plot(x, y, 'ro')
plt.show()
```

incrementing a Delaunay triangulation



Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

defining a sparse, weighted matrix

To introduce Dijkstra's algorithm to compute shortest paths, we define a row-based linked list sparse matrix (`lil_matrix`).

```
from scipy.sparse import lil_matrix
from scipy.sparse.csgraph import shortest_path

G = lil_matrix((4, 4))
G[0, 1] = 1
G[0, 2] = 2
G[1, 3] = 1
G[2, 0] = 2
G[2, 3] = 3
print(G)
```

calling Dijkstra's algorithm

```
dstmat, pred = shortest_path(csgraph=G,
    method='D', directed=False, indices=0,
    return_predecessors=True)

print('distances :', dstmat)
print('predecessors :', pred)
```

output of the script

```
   (0, 1) 1.0
   (0, 2) 2.0
   (1, 3) 1.0
   (2, 0) 2.0
   (2, 3) 3.0
distances : [0. 1. 2. 2.]
predecessors : [-9999     0     0     1]
```

Geometric Queries

1. Geometric Data Structures
   - problems and software
   - binary space partitions: k-d trees
   - searching in Delaunay triangulations and Voronoi diagrams
   - dynamic problems

2. Graphs, Distances, and Shortest Paths
   - Dijkstra's algorithm to compute shortest paths
   - computing a shortest path in the Lake Superior polygon

computing shortest paths

Following the Mastering Scipy book of Francisco J. Blanco-Silva, Packt Publishing 2015, we take the Lake Superior polygon.

A constrained conforming Delaunay triangulation (cfdt) is computed with the flag p to indicate that the source is a planar straight line graph and the flag D to enforce Steiner points.

We furthermore require that all triangles have a minimum angle of at least 20 degrees and impose a maximum area on triangles.

Consider the problem of computing the shortest path.

computing the Delaunay triangulation

The script starts with all imports, followed by the construction of the Delaunay triangulation.

```
from read_poly import read_poly
from matplotlib import pyplot as plt
from triangle import triangulate
from triangle import plot as tplot
from scipy.spatial import minkowski_distance
from scipy.sparse import lil_matrix
from scipy.sparse.csgraph import shortest_path

lake_superior = read_poly('superior.poly')
cncfq20adt = triangulate(lake_superior, 'pq20a.001D')
```

script continued, computing lengths

We collect indices of the vertices of all segments in the triangulation, and the lengths of these segments:

```
X = cncfq20adt['triangles'][:,0]
Y = cncfq20adt['triangles'][:,1]
Z = cncfq20adt['triangles'][:,2]
Xvert = [cncfq20adt['vertices'][x] for x in X]
Yvert = [cncfq20adt['vertices'][y] for y in Y]
Zvert = [cncfq20adt['vertices'][z] for z in Z]
lengthsXY = minkowski_distance(Xvert, Yvert)
lengthsXZ = minkowski_distance(Xvert, Zvert)
lengthsYZ = minkowski_distance(Yvert, Zvert)
```

script continued, running Dijkstra's algorithm

We create the weighted-adjacency matrix, stored as a `lil_matrix`, and compute the shortest path between the requested vertices:

```
nvert = len(cncfq20adt['vertices'])
G = lil_matrix((nvert, nvert))
for k in range(len(X)):
    G[X[k], Y[k]] = G[Y[k], X[k]] = lengthsXY[k]
    G[X[k], Z[k]] = G[Z[k], X[k]] = lengthsXZ[k]
    G[Y[k], Z[k]] = G[Z[k], Y[k]] = lengthsYZ[k]
```

and then call Dijkstra's algorithm to compute the shortest path.

```
dist_mat, pred = shortest_path(G, directed=True,
    method='D', return_predecessors=True)
```

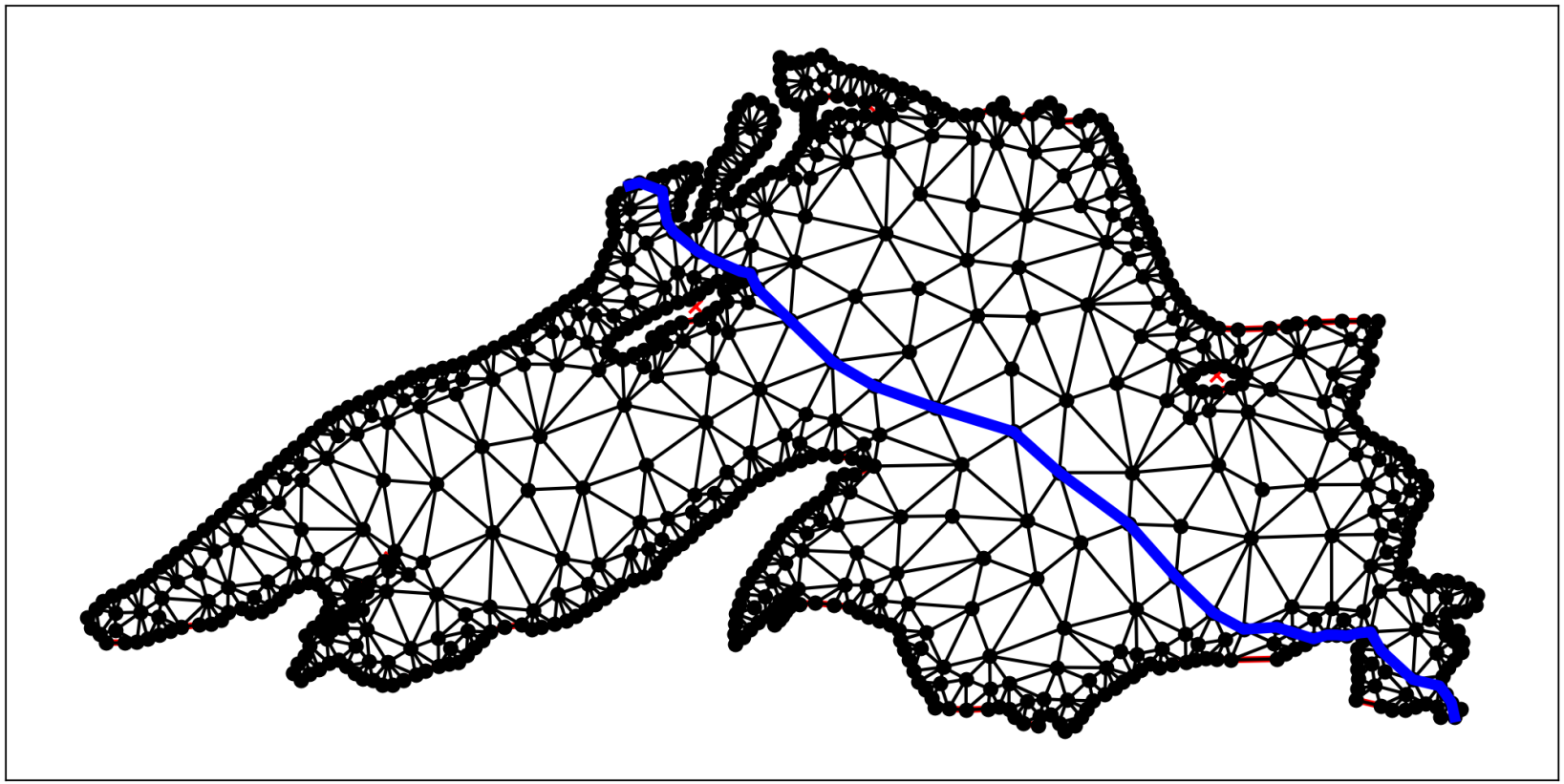
script continued, collect path and plot

We collect the vertices on the path and plot:

```
index = 370
path = [370]
while index != 197:
    index = pred[197, index]
    path.append(index)
print(path)

ax = plt.subplot(111)
tplot(ax, **cncfq20adt)
Xs = [cncfq20adt['vertices'][x][0] for x in path]
Ys = [cncfq20adt['vertices'][x][1] for x in path]
ax.plot(Xs, Ys, '-', linewidth=5, color='blue')
plt.show()
```

the shortest path



Exercises

1. Consider the code that displayed the less and the greater nodes, along with their children. Write a recursive function that displays the entire k-d tree, including all leaves of the tree.
2. For a 2-dimensional k-d tree, write a script to plot the k-d tree.