CMSC401
HW- 4
Mano Battula
118546490

1)
1.1 How is a raster terrain model defined? How can you ensure a continuous terrain approximation if
your data points are given at the vertices of a regular grid? Explain your answer.

Ans:
A raster terrain model is a grid-based digital representation of the terrain that includes information on the topography at each of the locations that the grid's cells represent on the surface of the Earth. Each cell in the grid stores a single number that reflects the elevation at that specific point. The height of the landscape is often depicted as a continuous surface.

When data points are provided at the vertices of a regular grid, many approaches, such as interpolation, resampling, and filtering, can be used to guarantee a continuous terrain approximation.

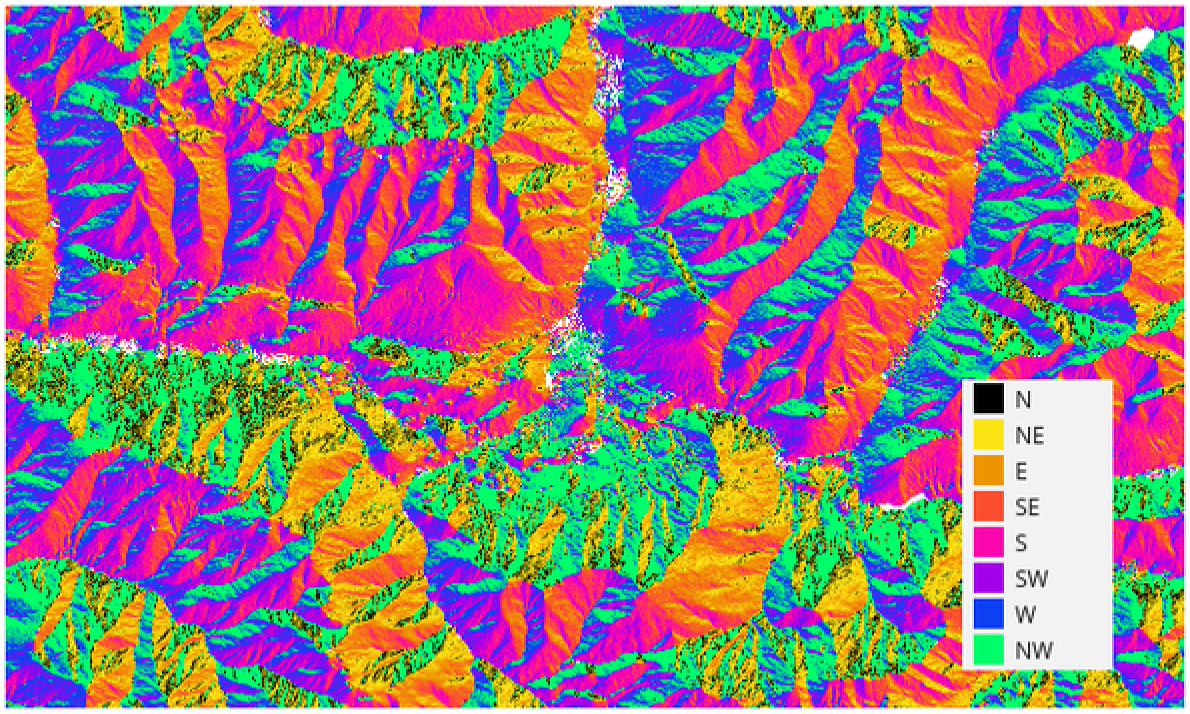
Interpolation is the process of extrapolating data from known values at nearby places to unknown locations. Bilinear interpolation is a popular interpolation technique that determines the elevation of a point inside a cell by calculating the difference between the heights at the cell's four corners.

By averaging or choosing values from the original grid to build a new grid at a different resolution, resampling includes transforming data from one grid resolution to another. This procedure can assist build a more continuous approximation of the landscape and can help smooth out angular terrain characteristics.

By applying mathematical procedures to the data, filtering removes any noise or inconsistencies in the landscape model. Low-pass filters and median filters are two common filtering methods that can assist reduce high-frequency noise while keeping the general form of the landscape.

In general, to produce a smooth and realistic depiction of the landscape, the right interpolation, resampling, and filtering procedures should be carefully chosen.

Example:



1.2) What are the characteristics of Regular Square Grids (RSGs) and Triangulated Irregular Networks
(TINs)? Discuss advantages and disadvantages of the two models.

Ans:
Regular Square Grids (RSGs) and Triangulated Irregular Networks (TINs) are two common models used to represent elevation data in digital terrain models.
RSGs are grids made up of square cells that are consistently spaced apart, each of which represents a point on the surface of the Earth and contains details about the topography there. Each cell in the grid stores a single number that reflects the elevation at that specific point. The height of the landscape is often depicted as a continuous surface.

Contrarily, TINs depict the landscape as a collection of asymmetrical triangles, with each triangle denoting a different area of the terrain's surface. Triangular faces that may be used to interpolate elevation values at any place inside the TIN are connected to a collection of dispersed elevation data points to produce TINs.

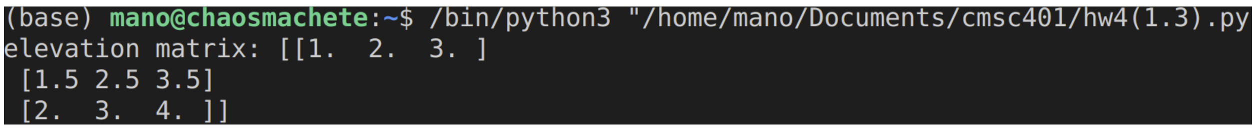
RSGs have several benefits, including their regular structure, ease of usage, and simplicity, which makes them suitable for a wide range of applications. They may be simply modified using common raster GIS tools and are particularly helpful for modeling terrain with constant elevation data, such as flat or moderately sloping ground. RSGs are also computationally effective, making it possible to handle and visualize big datasets quickly.

RSGs, however, may have several drawbacks. Particularly in places with significant elevational change, such mountainous areas, they frequently perform less accurately than other models. Additionally, they have a set grid resolution, which may cause the smoothing or loss of detail of landscape features. Finally, RSGs can be memory-intensive, particularly for large datasets with high resolution, requiring substantial storage and processing resources.
On the other hand, TINs provide a more adaptable representation of the landscape, particularly in regions with significant elevational variations. They are quickly updated or improved as new data becomes available, and they can properly depict complicated terrain characteristics like ridges, valleys, and cliffs. Because TINs are so flexible, it is possible to create irregular or uneven triangles that more accurately represent the contours of the ground.

TINs, however, can be computationally expensive, especially for huge datasets or extremely detailed models, necessitating a significant amount of computing power and storage. In addition, compared to RSGs, they need more specialized tools and knowledge to edit and evaluate. Lastly, TINs need rigorous data management and quality control since interpolation mistakes or inaccuracies might occur in areas with sparse or irregular data.
In general, the unique application and the characteristics of the terrain being represented will determine whether to use RSGs or TINs. Due to its simplicity, usability, and computing efficiency, RSGs are frequently a viable option for basic, generally flat terrain with reliable elevation data. TINs provide a more precise and adaptable representation for more complicated terrain with significant elevational fluctuation, but at the expense of greater processing complexity and specialist knowledge.

1.3)
Ans:

```
import numpy as np

def point_in_triangl (p, a, b, c):
    a, b, c = a[:2], b[:2], c[:2]
# Only use (x, y) coordinate
s   u = b - a
    v = c - a
    w = p - a
    v_cross_w = np.cross(v, w)
    v_cross_u = np.cross(v, u)
    if np.dot(v_cross_w, v_cross_u) < 0:
        return False

    u_cross_w = np.cross(u, w)
    u_cross_v = np.cross(u, v)
    if np.dot(u_cross_w, u_cross_v) < 0:
        return False

    denom = np.linalg.norm(u_cross_v)
    r = np.linalg.norm(v_cross_w) / denom
    t = np.linalg.norm(u_cross_w) / denom

    return r + t <= 1


def find_containing_triangl (p, triangles):
    for t in triangles:
        if point_in_triangl (p, t[0], t[1], t[2]):
           ereturn t
    return None

def interpolate_elevatio (p, triangle):
    a, b, c = triangle
    alpha = ((b[1] - c[1]) * (p[0] - c[0]) + (c[0] - b[0]) * (p
[1] - c[1])) / ((b[1] - c[1]) * (a[0] - c[0]) + (c[0] - b[0
]) * (a[1] - c[1]))
    beta = ((c[1] - a[1]) * (p[0] - c[0]) + (a[0] - c[0]) * (p[
1] - c[1])) / ((b[1] - c[1]) * (a[0] - c[0]) + (c[0] - b[0
]) * (a[1] - c[1]))
    gamma = 1 - alpha - beta
    zp = alpha * a[2] + beta * b[2] + gamma * c[2]
    return zp

def compute_elevation_matri (S, T, x1, y1, n, d):
    Z = np.zeros((n, n))
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            xi = x1 + (i - 1) * d
            yj = y1 + (j - 1) * d
```


```
(base) mano@chaosmachete:~$ /bin/python3 "/home/mano/Documents/cmsc401/hw4(1.3).py
elevation matrix: [[1.  2.  3. ]
 [1.5 2.5 3.5]
 [2.  3.  4. ]]
```

2)
2.1)Describe the indexed data structure with adjacencies (IA data structure) for a triangle mesh, and discuss its implementation and storage cost.

Ans:
 Triangle meshes are frequently represented using the Indexed Adjacency (IA) data structure, which effectively holds data on vertices, triangles, and their adjacency relationships. The IA data structure is made up of three parts:

Vertex list: a list of the special vertices in the mesh. Each vertex can be represented as a tuple or array of either 2D (x, y) or 3D (x, y, z) coordinates.

Triangle list: A list of triangles in a mesh, each represented by a tuple or array of three indices which are identical to the vertex list's indices.

Adjacency list: A list that maintains information about the adjacency connections between triangles. For every triangle, the adjacency list stores the indices of its neighboring triangles that share an edge. A triangle can have a maximum of three neighboring triangles.

The IA data structure is not only storage-efficient but also allows rapid access to vertex and triangle data. Its storage cost can be broken down as follows:

Vertex list: The mesh's total number of vertices, V, is used to calculate the storage cost. Each vertex needs room to store its (x, y) or (x, y, z) coordinates as well as any other properties, such as color, texture coordinates, or normals, if applicable.

Triangle list: A list of triangles in a mesh, each represented by a tuple or array of three indices that are identical to the vertex list's indices.

Adjacency list: The storage cost is O(T) since each triangle can have up to three neighboring triangles, and their indices need to be stored.

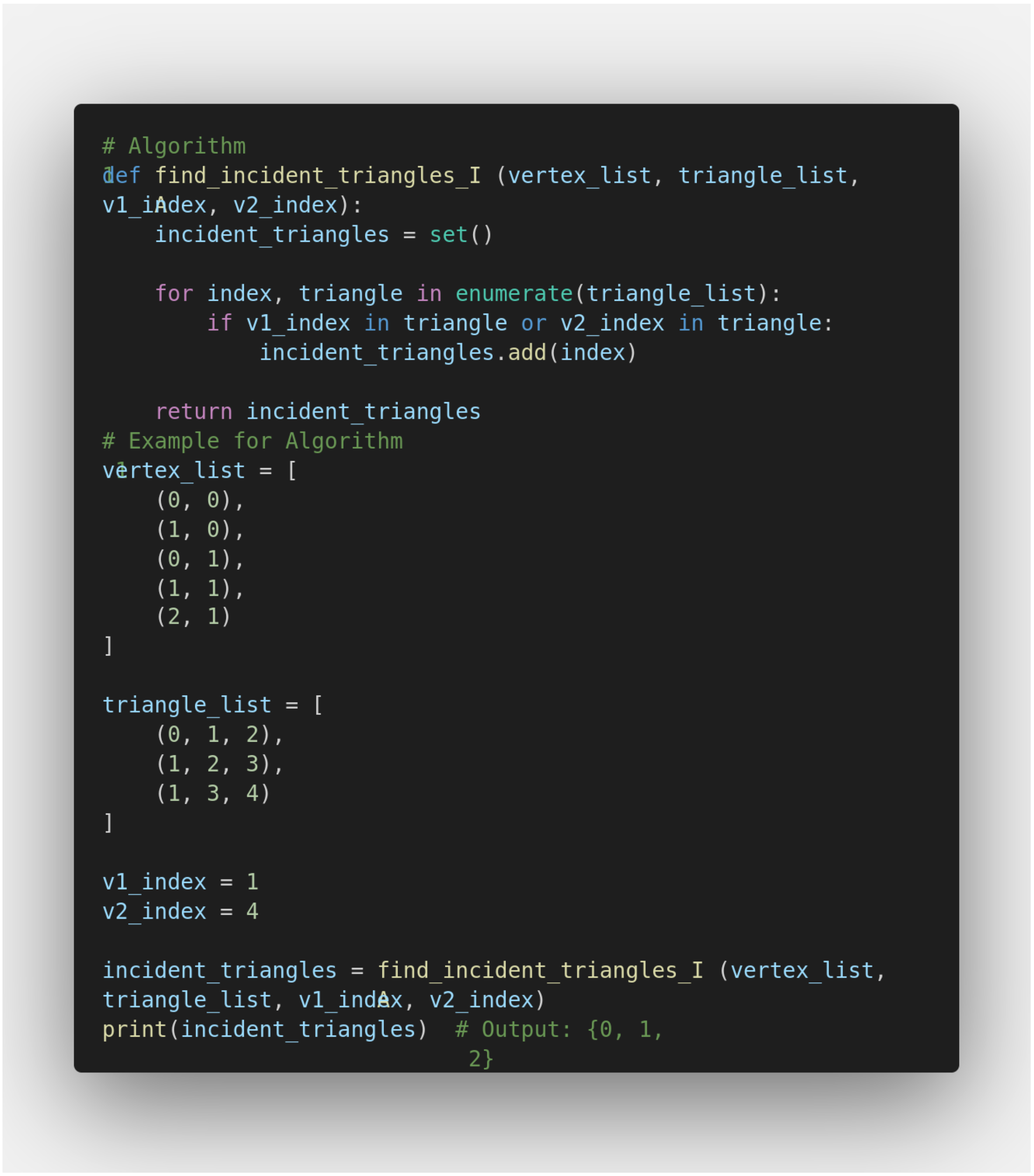
Overall, the IA data structure's storage cost is O(V + 2T), which is typically dominated by the number of vertices and triangles in the mesh.

The IA data structure can be implemented using arrays, lists, or dictionaries in most programming languages. Arrays or lists enable efficient access to the corresponding data of vertex and triangle indices. For adjacency information, a 2D array or list can be utilized, with the first dimension corresponding to the triangle index and the second dimension containing the neighboring triangle indices. Alternatively, a dictionary can map triangle indices to their neighboring triangle indices, offering more flexibility for complex meshes with varying numbers of neighbors per triangle.

2.2)
Given an internal edge of a triangle mesh joining two vertices v1 and v2 , describe two algorithms for computing the list of the triangles which are incident either in v1 or v2 (or in both) without duplicates:
• the first algorithm needs to be based on an encoding of the mesh as an IA data structure;
• the second algorithm needs to be based on an encoding of the mesh as a Triangle PR-quadtree,
under the assumption that v1 and v2 belong to the same quadtree leaf block.

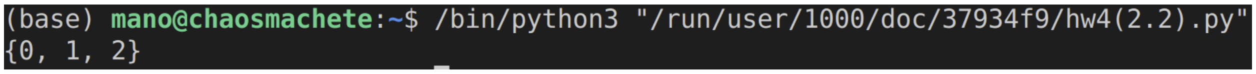
Ans:

Algorithm1

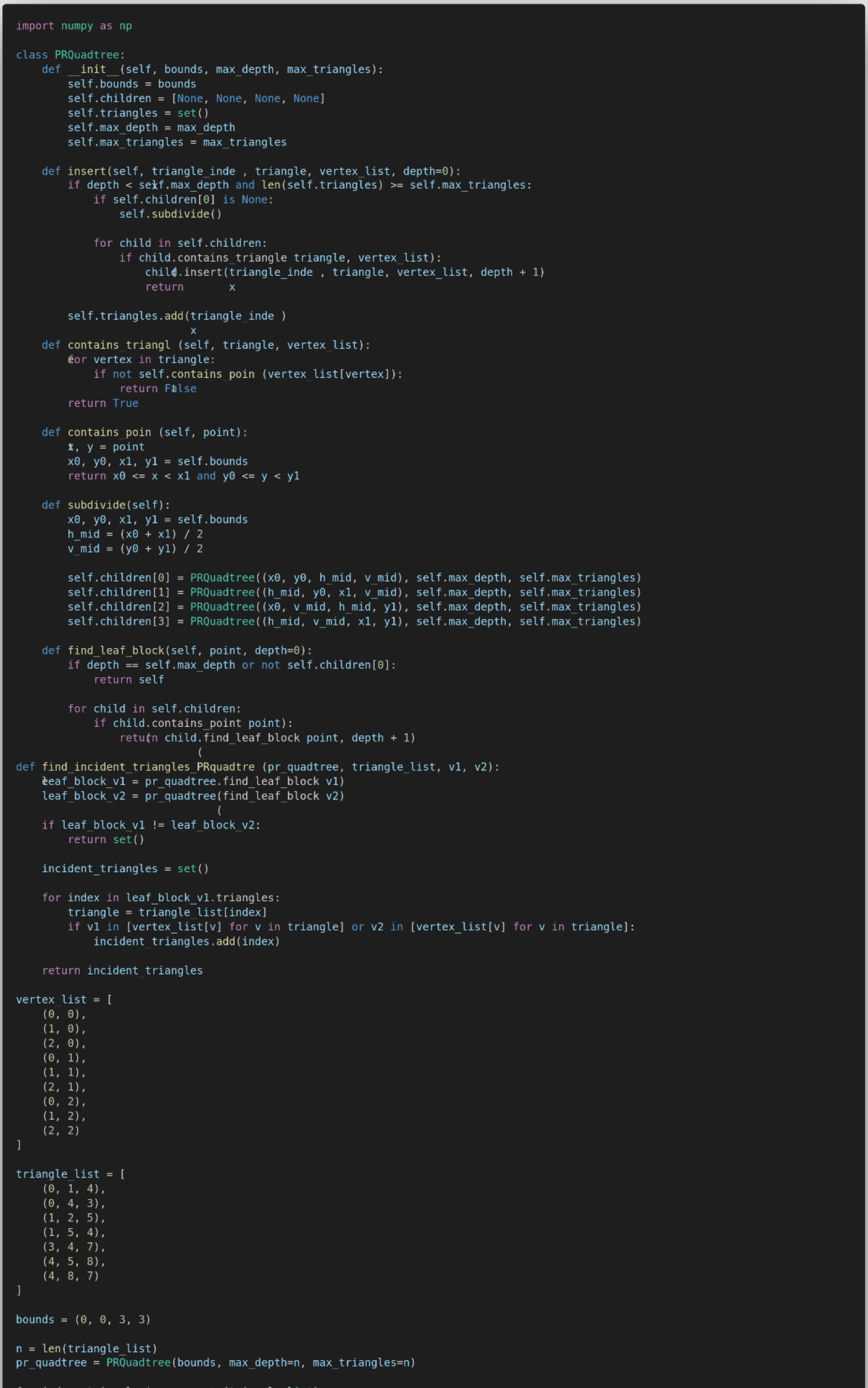


```
# Algorithm
def find_incident_triangles_I (vertex_list, triangle_list,
v1_index, v2_index):
    incident_triangles = set()

    for index, triangle in enumerate(triangle_list):
        if v1_index in triangle or v2_index in triangle:
            incident_triangles.add(index)

    return incident_triangles
# Example for Algorithm
vertex_list = [
    (0, 0),
    (1, 0),
    (0, 1),
    (1, 1),
    (2, 1)
]

triangle_list = [
    (0, 1, 2),
    (1, 2, 3),
    (1, 3, 4)
]

v1_index = 1
v2_index = 4

incident_triangles = find_incident_triangles_I (vertex_list,
triangle_list, v1_index, v2_index)
print(incident_triangles)  # Output: {0, 1,
                             2}
```

OUTPUT:

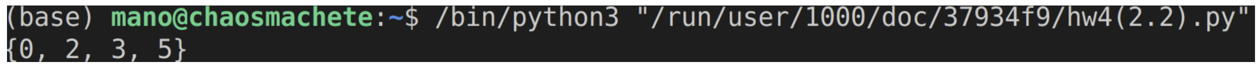


```
(base) mano@chaosmachete:~$ /bin/python3 "/run/user/1000/doc/37934f9/hw4(2.2).py"
{0, 1, 2}
```



```python
import numpy as np

class PRQuadtree:
    def __init__(self, bounds, max_depth, max_triangles):
        self.bounds = bounds
        self.children = [None, None, None, None]
        self.triangles = set()
        self.max_depth = max_depth
        self.max_triangles = max_triangles

    def insert(self, triangle_inde , triangle, vertex_list, depth=0):
        if depth < self.max_depth and len(self.triangles) >= self.max_triangles:
            if self.children[0] is None:
                self.subdivide()

            for child in self.children:
                if child.contains_triangle triangle, vertex_list):
                    child.insert(triangle_inde , triangle, vertex_list, depth + 1)
                    return       x

        self.triangles.add(triangle_inde )
                           x
    def contains_triangl (self, triangle, vertex_list):
        for vertex in triangle:
            if not self.contains_poin (vertex_list[vertex]):
                return False
        return True

    def contains_poin (self, point):
        x, y = point
        x0, y0, x1, y1 = self.bounds
        return x0 <= x < x1 and y0 <= y < y1

    def subdivide(self):
        x0, y0, x1, y1 = self.bounds
        h_mid = (x0 + x1) / 2
        v_mid = (y0 + y1) / 2

        self.children[0] = PRQuadtree((x0, y0, h_mid, v_mid), self.max_depth, self.max_triangles)
        self.children[1] = PRQuadtree((h_mid, y0, x1, v_mid), self.max_depth, self.max_triangles)
        self.children[2] = PRQuadtree((x0, v_mid, h_mid, y1), self.max_depth, self.max_triangles)
        self.children[3] = PRQuadtree((h_mid, v_mid, x1, y1), self.max_depth, self.max_triangles)

    def find_leaf_block(self, point, depth=0):
        if depth == self.max_depth or not self.children[0]:
            return self

        for child in self.children:
            if child.contains_point point):
                return child.find_leaf_block point, depth + 1)
                             (
def find_incident_triangles_PRquadtre (pr_quadtree, triangle_list, v1, v2):
    leaf_block_v1 = pr_quadtree.find_leaf_block v1)
    leaf_block_v2 = pr_quadtree(find_leaf_block v2)
                               (
    if leaf_block_v1 != leaf_block_v2:
        return set()

    incident_triangles = set()

    for index in leaf_block_v1.triangles:
        triangle = triangle_list[index]
        if v1 in [vertex_list[v] for v in triangle] or v2 in [vertex_list[v] for v in triangle]:
            incident_triangles.add(index)

    return incident_triangles

vertex_list = [
    (0, 0),
    (1, 0),
    (2, 0),
    (0, 1),
    (1, 1),
    (2, 1),
    (0, 2),
    (1, 2),
    (2, 2)
]

triangle_list = [
    (0, 1, 4),
    (0, 4, 3),
    (1, 2, 5),
    (1, 5, 4),
    (3, 4, 7),
    (4, 5, 8),
    (4, 8, 7)
]

bounds = (0, 0, 3, 3)

n = len(triangle_list)
pr_quadtree = PRQuadtree(bounds, max_depth=n, max_triangles=n)
```

Algorithm 2 OUTPUT:



```
(base) mano@chaosmachete:~$ /bin/python3 "/run/user/1000/doc/37934f9/hw4(2.2).py"
{0, 2, 3, 5}
```

3)

3.1) What are the characteristics that make a Delaunay triangulation suitable as the basis for a TIN
representing a terrain?

Ans:

Delaunay triangulation is a popular choice for creating Triangulated Irregular Networks (TINs) when representing terrains. The reasons for its suitability include the following attributes:

- **Improved triangle quality:** The Delaunay triangulation method aims to optimize the minimum angles of the triangles, preventing the formation of excessively elongated or narrow triangles that may cause inaccuracies in the terrain representation.

- **No vertices within circumcircles:** A key characteristic of Delaunay triangulation is that no vertex is located within the circumcircle of any other triangle. This ensures an even distribution of triangles and reduces the chances of overlapping triangles in the terrain model.

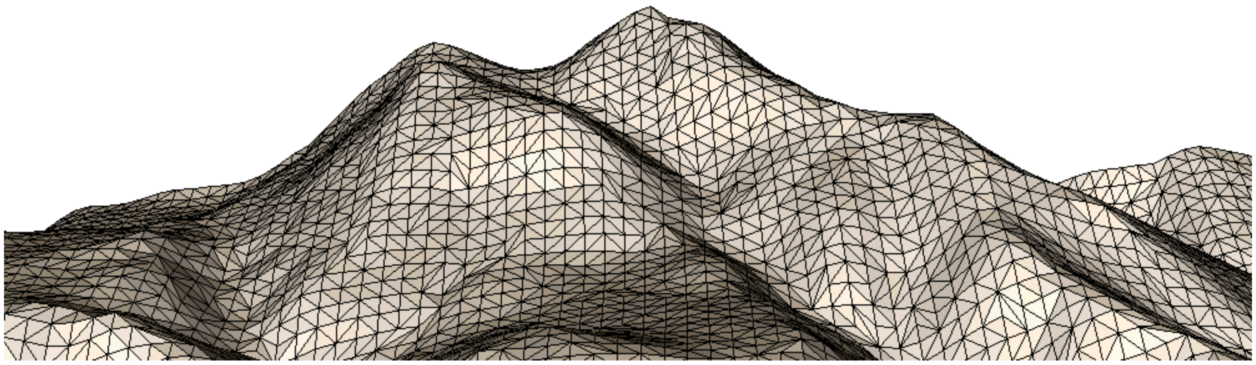
- **Flexible sampling:** Delaunay triangulation naturally adjusts to different point densities, effectively handling terrains with varying levels of detail. In high-density regions, the triangulation generates smaller triangles, while larger triangles are created in areas with fewer points. This flexibility leads to a more efficient terrain representation.

- **Consistency:** For input points in general position (with no four points lying on a circle), the Delaunay triangulation is unique. This consistency simplifies the implementation and analysis of algorithms that work with TINs.

- **Elevation interpolation:** Delaunay triangulation enables natural interpolation of elevation values between vertices. This allows for the estimation of elevation values at any location within the terrain by interpolating the values of the vertices that make up the surrounding triangle.

- **Hydrological modeling support:** Delaunay triangulation helps maintain the correct drainage patterns of the terrain, as water flow follows the paths determined by both the elevation values and the triangulation.

In conclusion, the Delaunay triangulation approach is well-suited for TIN-based terrain representation due to its ability to accommodate varying point densities, generate well-shaped triangles, ensure unique and evenly distributed triangles, and support hydrological modeling.
Example:

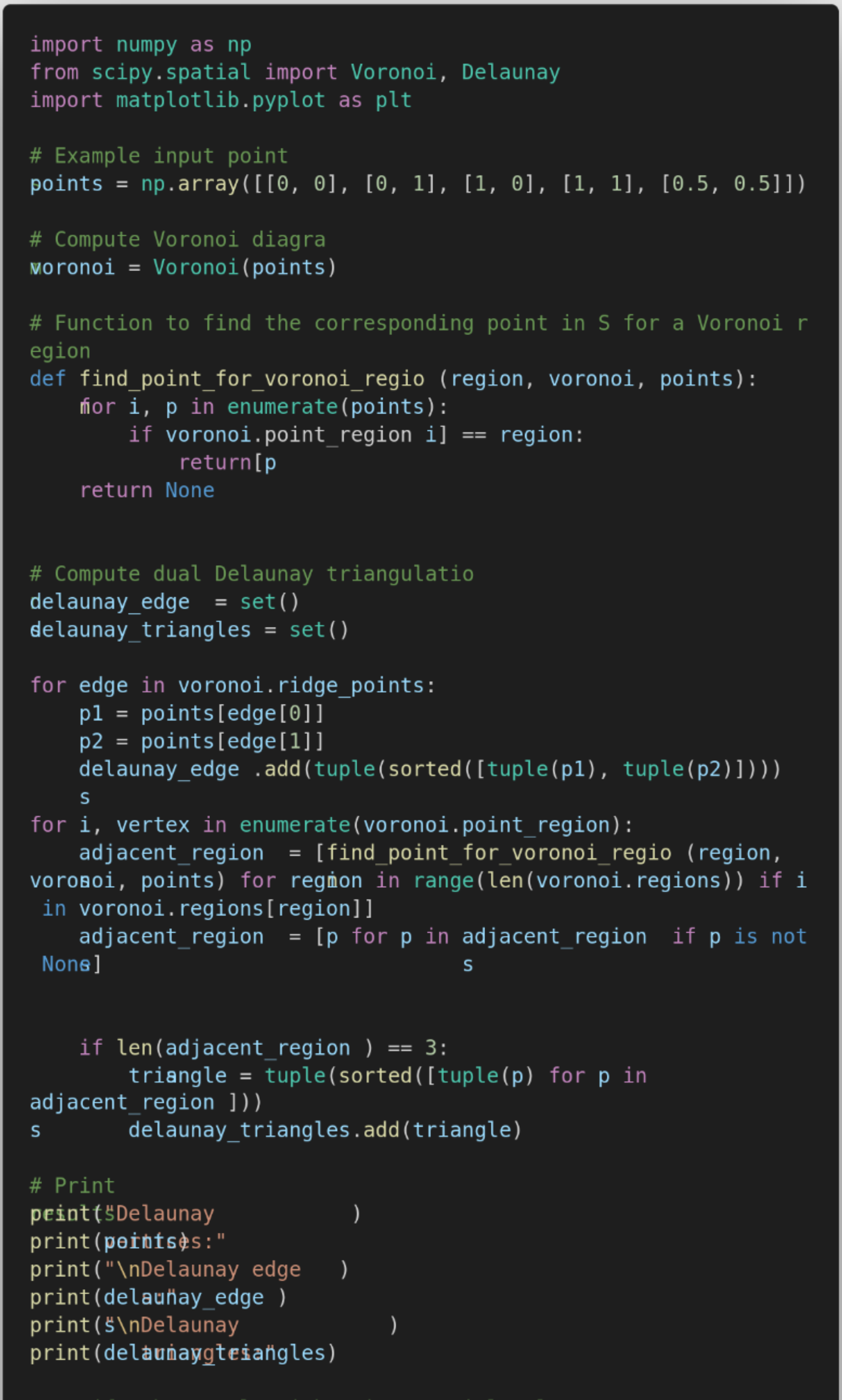


3.2) When is a Delaunay triangulation of a set of points not unique? Explain your answer.
Ans:
A Delaunay triangulation for a given set of points might not be unique when there are several possible triangulations that adhere to the Delaunay condition. This condition specifies that no point should be positioned inside the circumcircle of any triangle. The non-uniqueness generally arises when the point set contains four or more cocircular points, which means they all share the same circle.

In instances like this, the edges connecting the cocircular points shape a convex quadrilateral that has interchangeable diagonals. Both potential triangulations of this quadrilateral satisfy the Delaunay condition because neither diagonal infringes upon the rule that no point should be present within any triangle's circumcircle.

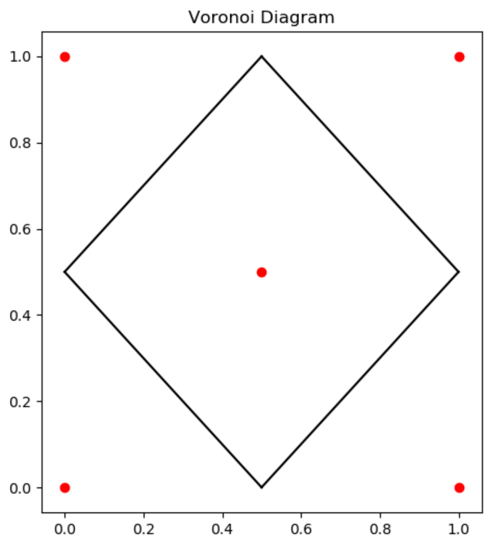
Therefore, when a point set contains four or more cocircular points, multiple valid Delaunay triangulations can be generated, leading to non-uniqueness. However, it is important to note that this occurrence is relatively uncommon, and Delaunay triangulations are typically unique for the majority of point sets that are in general position.

3.3)
Ans:

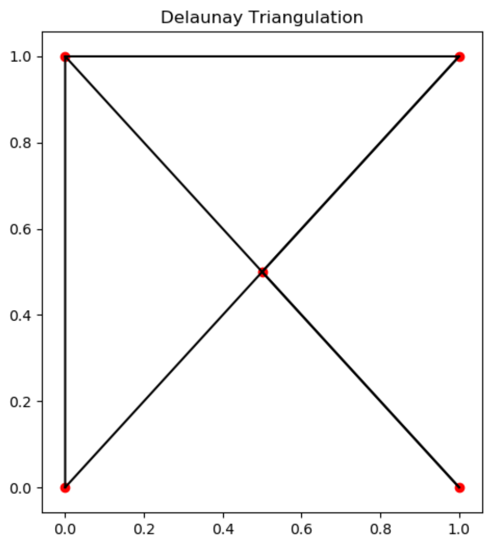


```python
import numpy as np
from scipy.spatial import Voronoi, Delaunay
import matplotlib.pyplot as plt

# Example input point
points = np.array([[0, 0], [0, 1], [1, 0], [1, 1], [0.5, 0.5]])

# Compute Voronoi diagra
voronoi = Voronoi(points)

# Function to find the corresponding point in S for a Voronoi r
egion
def find_point_for_voronoi_regio (region, voronoi, points):
    for i, p in enumerate(points):
        if voronoi.point_region i] == region:
            return[p
    return None


# Compute dual Delaunay triangulatio
delaunay_edge  = set()
delaunay_triangles = set()

for edge in voronoi.ridge_points:
    p1 = points[edge[0]]
    p2 = points[edge[1]]
    delaunay_edge .add(tuple(sorted([tuple(p1), tuple(p2)])))
    s
for i, vertex in enumerate(voronoi.point_region):
    adjacent_region  = [find_point_for_voronoi_regio (region,
voronoi, points) for region in range(len(voronoi.regions)) if i
 in voronoi.regions[region]]
    adjacent_region  = [p for p in adjacent_region  if p is not
 None]                                s


    if len(adjacent_region ) == 3:
        triangle = tuple(sorted([tuple(p) for p in
adjacent_region ]))
s       delaunay_triangles.add(triangle)

# Print
print("Delaunay points:"           )
print(points)
print("\nDelaunay edge    )
print(delaunay_edge )
print("\nDelaunay triangles:"         )
print(delaunay_triangles)
```

OUTPUT:



Voronoi Diagram
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0



Delaunay Triangulation
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0