Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

Факультет «Информатика и системы управления»

Кафедра «Теоретическая информатика и компьютерные технологии»

Домашняя работа № 1

по курсу «Моделирование»

ПОСТРОЕНИЕ СТАТИСТИЧЕСКОЙ МОДЕЛИ ПО РЕЗУЛЬТАТАМ ДИСТАНЦИОННОГО ЗОНДИРОВАНИЯ ЗЕМЛИ

Студент: Яровикова А. С.

Группа: ИУ9-81Б

Преподаватель: Домрачева А. Б.

Москва, 2024

ЦЕЛЬ И ПОСТАНОВКА ЗАДАЧИ

Цель

Целью данной работы является изучение и реализация триангуляции Делоне для построения статистической модели.

Постановка задачи

Построить цифровую модель рельефа местности по представленному космическому снимку (гора Эльбрус). Представить визуализацию без текстурирования и закрашивания с возможностью выбора точки и оценки значения высоты в этой точке.

Для реализации цифровой модели используется триангуляция Делоне с помощью итеративного алгоритма «Удаляй и строй».

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Задача построения триангуляции Делоне является одной из базовых в вычислительной геометрии. Она широко используется в геоинформационных системах для моделирования поверхностей и решения пространственных задач.

Триангуляцией называется планарный граф, все внутренние области которого являются треугольниками.

Выпуклой триангуляцией называется такая триангуляция, для которой минимальный многоугольник, охватывающий все треугольники, будет выпуклым. Триангуляция, не являющаяся выпуклой, называется невыпуклой.

Триангуляция **удовлетворяет условию Делоне**, если внутрь окружности, описанной вокруг любого построенного треугольника, не попадает ни одна из заданных точек триангуляции.

Триангуляцией Делоне называется триангуляция, которая является выпуклой и удовлетворяет условию Делоне. Пример такой триангуляции представлен на рисунке 1.

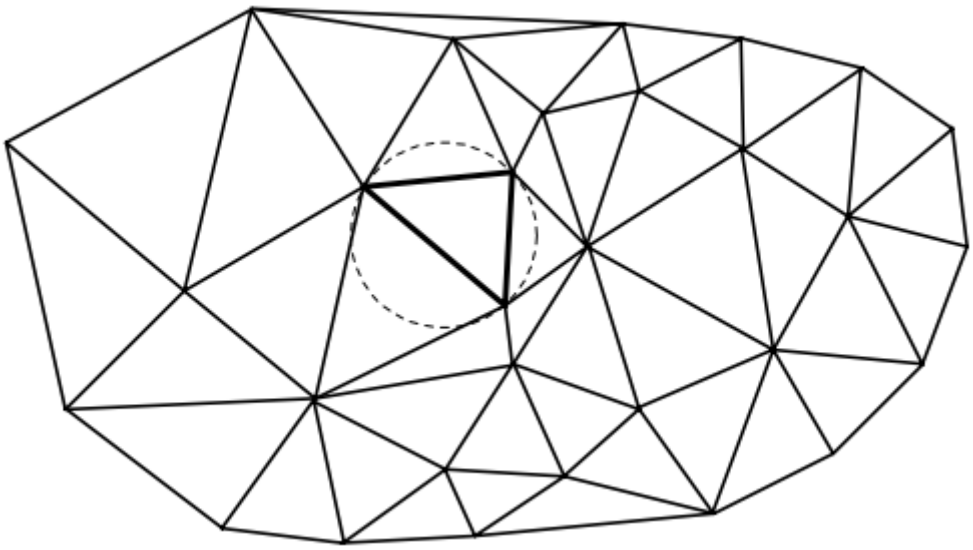

Рисунок 1 - Триангуляция Делоне

Итеративный алгоритм «Удаляй и строй»

Алгоритм основывается на последовательном добавлении точек в частично построенную триангуляцию.

При каждом добавлении нового узла происходит удаление всех треугольников, в которых новый узел находится внутри описанных окружностей. Эти удаленные треугольники вместе формируют некоторый многоугольник, который неявно определен. После удаления треугольников на месте возникшего многоугольника осуществляется создание заполняющей триангуляции путем соединения нового узла с этим многоугольником. Последовательное выполнение этих шагов алгоритма представлено на рисунке 2.

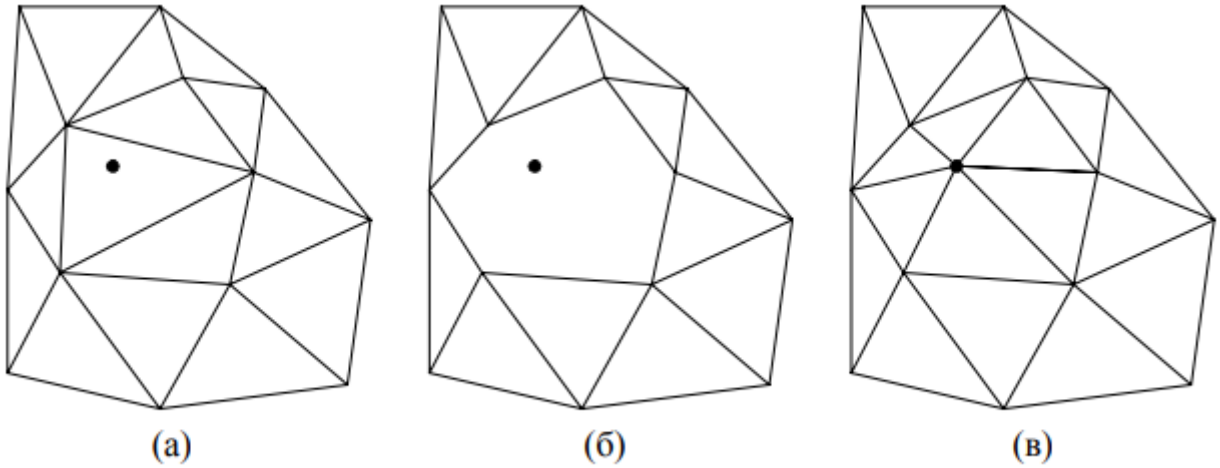

Рисунок 2 - Вставка точки в итеративном алгоритме «Удаляй и строй»: а – локализация точки в треугольнике; б – удаление треугольников; в – построение новых треугольников

Данный алгоритм строит сразу все необходимые треугольники в отличие от обычного итеративного алгоритма, где при вставке одного узла возможны многократные перестроения одного и того же треугольника. Однако здесь на первый план выходит процедура выделения контура удаленного многоугольника, от эффективности работы которого зависит общая скорость алгоритма. В целом в зависимости от используемой структуры данных этот алгоритм может тратить времени меньше, чем алгоритм с перестроениями, и наоборот.

ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ

Исходный код, реализующий построение статистической модели на основе триангуляции Делоне, представлен ниже.

Импортирование необходимых библиотек:

```
import numpy as np
import matplotlib.pyplot as plt
import math
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from PIL import Image
```

Класс точки в двумерном пространстве:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return "Point(" + str(self.x) + ", " + str(self.y) + ")"

    def __hash__(self):
        return hash(str(self))

    def __eq__(self, other):
        return isinstance(other, self.__class__) and
self.x == getattr(other, "x", None) and self.y == getattr(other, "y", None)

    def __ne__(self, other):
        return not (self == other)

    def get_dist_to_point(self, point):
        return math.sqrt(pow(self.x - point.x, 2) + pow(self.y - point.y, 2))
```

Класс отрезка, соединяющего точки в двумерном пространстве:

```
class Edge:
    def __init__(self, p1, p2):
        self.p1, self.p2 = p1, p2

    def __str__(self):
        return "Edge(" + str(self.p1) + ", " + str(self.p2) + ")"

    def __hash__(self):
        return hash(hash(str(self.p1)) + hash(str(self.p2)))

    def __eq__(self, other):
        return isinstance(other, self.__class__) and \
            (self.p1 == getattr(other, "p1", None) and self.p2 == getattr(other,
"p2", None)) or \
            (self.p1 == getattr(other, "p2", None) and self.p2 == getattr(other,
"p1", None))

    def __ne__(self, other):
        return not (self == other)
```

Класс окружности в двумерном пространстве:

```
class Circle:
    def __init__(self, center_point, radius):
        self.center_point, self.radius = center_point, radius

    def contains_point(self, point):
        return point.get_dist_to_point(self.center_point) <= self.radius
```

Класс треугольника в двумерном пространстве:

```
class Triangle:
    def __init__(self, p1, p2, p3):
        self.p1, self.p2, self.p3 = p1, p2, p3

    def __str__(self):
        return "Triangle(" + str(self.p1) + ", " + str(self.p2) + ", " +
str(self.p3) + ")"

    def __eq__(self, other):
        return isinstance(other, self.__class__) and \
            self.p1 == getattr(other, "p1", None) and \
            self.p2 == getattr(other, "p2", None) and \
            self.p3 == getattr(other, "p3", None)

    def __ne__(self, other):
        return not (self == other)

    def get_points(self):
        return [self.p1, self.p2, self.p3]

    def get_edges(self):
        return [Edge(self.p1, self.p2), Edge(self.p2, self.p3), Edge(self.p3,
self.p1)]

    def get_circumscribed_circle(self):
        z = []
        for point in self.get_points():
            z.append(point.x * point.x + point.y * point.y)

        z_x = (self.p1.y - self.p2.y) * z[2] + (self.p2.y - self.p3.y) * z[0] +
(self.p3.y - self.p1.y) * z[1]
        z_y = (self.p1.x - self.p2.x) * z[2] + (self.p2.x - self.p3.x) * z[0] +
(self.p3.x - self.p1.x) * z[1]
        z = (self.p1.x - self.p2.x) * (self.p3.y - self.p1.y) - (self.p1.y -
self.p2.y) * (self.p3.x - self.p1.x)

        a = -z_x / (2 * z)
        b = z_y / (2 * z)
        point = Point(a, b)
        return Circle(point, point.get_dist_to_point(self.p1))

    def draw(self, ax):
        x = [self.p1[0], self.p2[0], self.p3[0]]
        y = [self.p1[1], self.p2[1], self.p3[1]]
        z = [self.p1[2], self.p2[2], self.p3[2]]
        vertices = [list(zip(x, y, z))]
        ax.add_collection3d(Poly3DCollection(vertices, edgecolors='r', lw=2,
alpha=0.2, linestyles='--'))
```

Функция триангуляции Делоне с реализацией итеративного алгоритма «Удаляй и строй»:

```
def delaunay_triangulation(source_points, dist):
    points = set(source_points)

    if len(points) == 0:
        return []

    # Находим минимальные и максимальные значения координат
    x_min, x_max, y_min, y_max = None, None, None, None
    for point in points:
        if x_min is None or x_min > point.x:
            x_min = point.x
        if x_max is None or x_max < point.x:
            x_max = point.x
        if y_min is None or y_min > point.y:
            y_min = point.y
        if y_max is None or y_max < point.y:
            y_max = point.y
    # Ограничивающая рамка расширяется путем добавления/вычитания значения dist
    x_min -= dist
    y_min -= dist
    x_max += dist
    y_max += dist

    # Создаем начальные треугольники
    # Два начальных треугольника созданы с использованием углов ограничивающей
рамки
    triangles = [
        Triangle(Point(x_min, y_max), Point(x_min, y_min), Point(x_max, y_min)),
        Triangle(Point(x_min, y_max), Point(x_max, y_max), Point(x_max, y_min)),
    ]

    # Добавляем новые точки по правилу удаляй и строй
    # Если новая точка попадает в окружность треугольника, треугольник
помечается как "плохой треугольник",
    # а его края помечаются как "плохие ребра". Плохие треугольники удаляются из
списка
    # Новые треугольники создаются путем соединения новой точки с ребрами плохих
треугольников, которые имеют только 1 вхождение в словаре плохих ребер (т.е. это
не смежное ребро)
    for new_point in points:
        bad_triangles = []
        bad_edges = {}
        for triangle in triangles:
            circle = triangle.get_circumscribed_circle()
            # Проверяем, лежит ли новая точка в окружности, описанной вокруг
треугольника
            if circle.contains_point(new_point):
                bad_triangles.append(triangle)
                # Запоминаем плохие ребра, которые будут удалены
                for bad_triangle_edge in triangle.get_edges():
                    bad_edges[bad_triangle_edge] =
bad_edges.get(bad_triangle_edge, 0) + 1

        # Удаляем плохие треугольники
        for bad_triangle in bad_triangles:
```

```
            triangles.remove(bad_triangle)

        # Строим новые треугольники
        for bad_edge in bad_edges:
            if bad_edges[bad_edge] > 1:
                continue
            new_triangle = Triangle(new_point, bad_edge.p1, bad_edge.p2)
            triangles.append(new_triangle)

    return triangles
```

Инициализация основных параметров (высота Эльбруса и количество точек для модели):

```
mountaint_height = 5642
point_count = 1500
dist = 1 # значение расстояния, используемое для расширения ограничивающей рамки
```

Сканирование и конвертация космического снимка горы Эльбрус:

```
im = Image.open('img1.jpg').convert('L').crop((700, 300, 1500, 1000))
im_array = np.array(im)
im_array = im_array.astype(float) / 255
width, height = im.size
```

Функция вычисления средней высоты в точке:

```
def avg_height(y, x):
    eps = 40
    return im_array[max(x - eps, 0):min(x + eps, height), max(y - eps, 0):min(y
+ eps, width)].mean()
```

Запуск программы:

```
source_points = []
for i in range(point_count):
    random_x = np.random.randint(0, width - 1)
    random_y = np.random.randint(0, height - 1)
    source_points.append(Point(random_x, random_y))

delaunay_triangles = delaunay_triangulation(source_points, dist)

triangles = []
for triangle in delaunay_triangles:
    p1, p2, p3 = triangle.p1, triangle.p2, triangle.p3
    z1, z2, z3 = avg_height(p1.x, p1.y) * mountaint_height, avg_height(p2.x,
p2.y) * mountaint_height, avg_height(p3.x, p3.y) * mountaint_height

    triangles.append([(p1.x, p2.x, p3.x), (p1.y, p2.y, p3.y), (z1, z2, z3)])

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

for triangle in triangles:
    color = np.mean(triangle[2])/5642.0
    ax.plot(*triangle, color=str(color), linewidth=1, gapcolor=str(color))

ax.set_xlabel('X')
```

```
ax.set_ylabel('Y')
ax.set_zlabel('Z')
# If we knew what angles we wanted to set, these lines will set it
elev = 50.0
azim = 90.5
ax.view_init(elev, azim)

# Show the figure, adjust it with the mouse
plt.show()
```

РЕЗУЛЬТАТЫ

Результаты построения цифровой модели рельефа местности по представленному космическому снимку горы Эльбрус представлены на рисунках ниже.

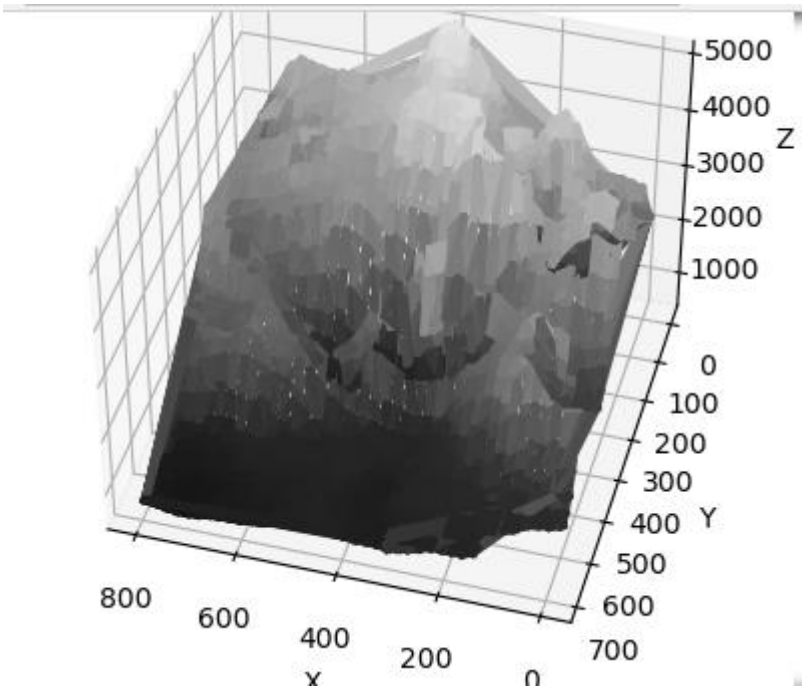

Рисунок 3 - Визуализация рельефа горы: толщина линий 3 пикселя

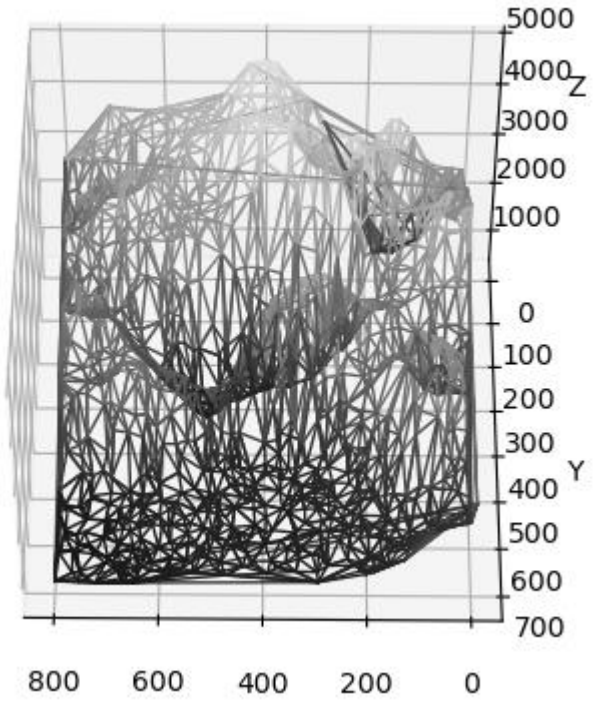

Рисунок 4 - визуализация рельефа горы: толщина линий 1 пиксель

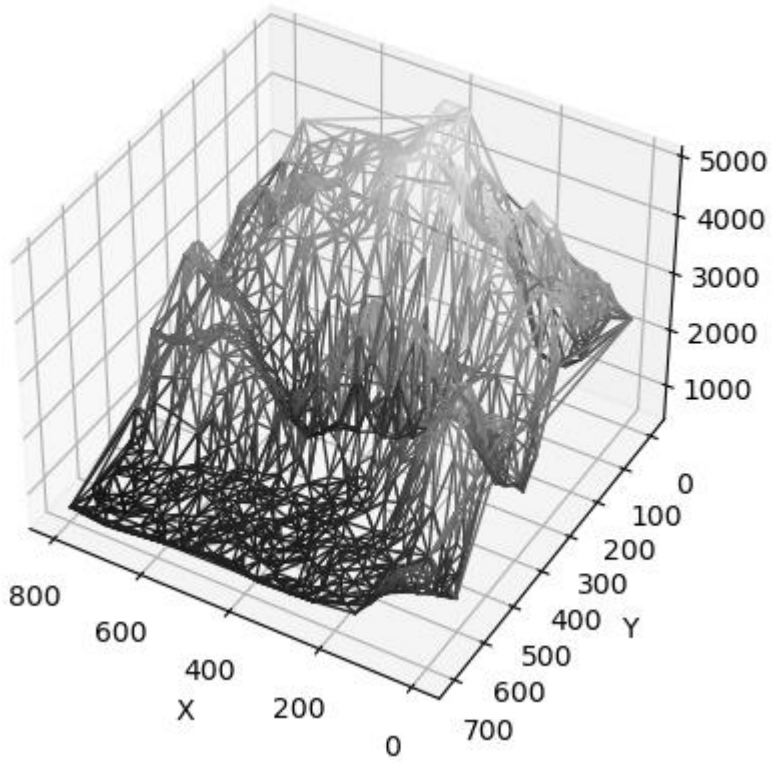

Рисунок 5 - Визуализация рельефа горы под углом

```
1700.4653419416547
```

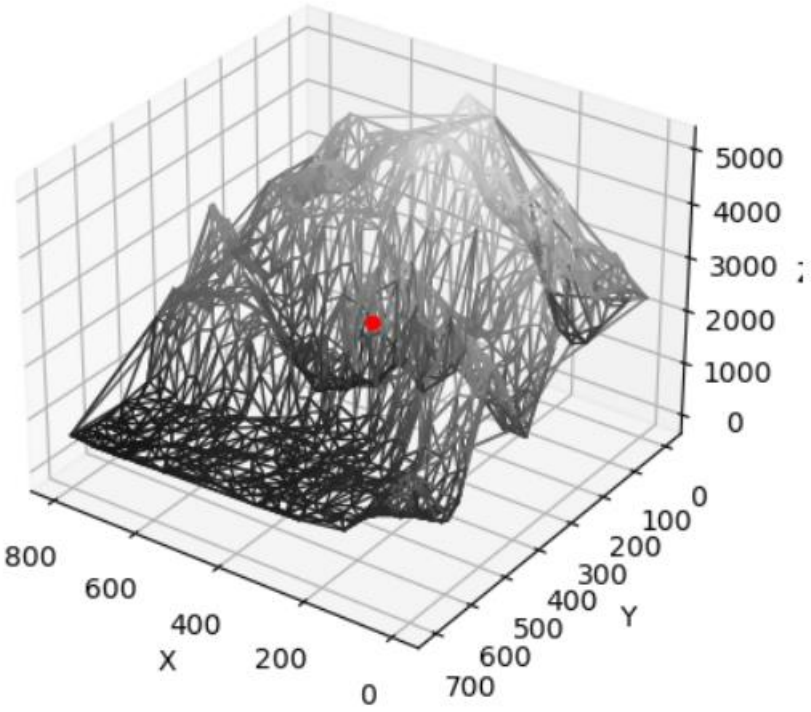

Рисунок 6 - Визуализация рельефа горы с выбранной точкой и найденной высотой

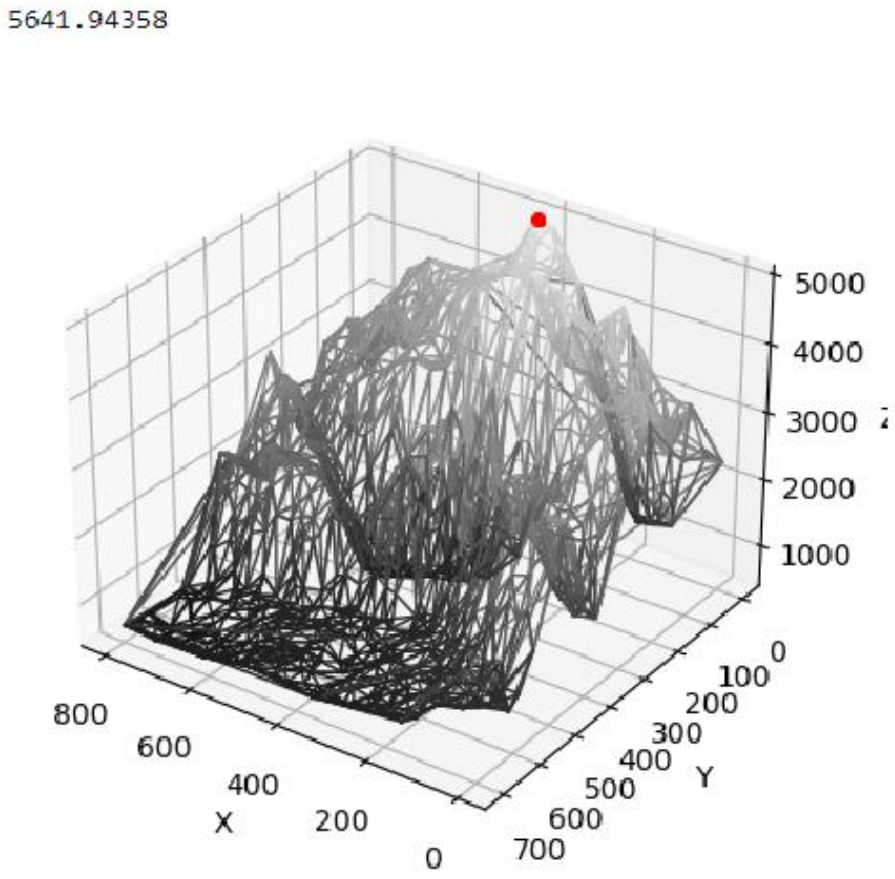

Рисунок 7 - Визуализация рельефа горы с выбранной точкой близ вершины и найденной высотой

ВЫВОДЫ

В ходе выполнения лабораторной работы был изучен метод триангуляции Делоне и реализован итеративный алгоритм «Удаляй и строй» на языке программирования Python. Была построена цифровая статистическая модель по результатам дистанционного зондирования Земли (космический снимок горы Эльбрус)

В итоге получились неплохие результаты: построенная модель визуально похожа на исходный снимок горы, а также функция подсчета средней высоты в точке функционирует исправно и возвращает корректные значения с небольшой погрешностью в 0.01. Однако результаты также позволили сделать вывод о том, что неточность значений высот связаны с рядом ключевых условий.

Важно отметить, что идеальное время для получения космических снимков подобного рода задач – это момент, когда солнце находится в зените. Это позволит избежать искусственных затенений и обеспечить наилучшее качество изображения.

Кроме того, метод триангуляции Делоне эффективнее проявляет свою работоспособность на равнинных участках или участках с небольшими перепадами высот. На гористых местностях, таких как гора Эльбрус, могут возникать искусственные тени от выступающих пиков, что может повлиять на точность моделирования и интерпретацию результатов.

Таким образом, хочется подчеркнуть практическую применимость метода триангуляции Делоне для анализа космических снимков рельефа Земли. При этом крайне важно учитывать особенности местности и условий освещения, что позволит получить наиболее точные и достоверные результаты моделирования при анализе рельефа местностей и других пространственных задач.