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»

Лабораторная работа №1

по курсу «Моделирование»

«Построение статической модели»

Студент группы ИУ9-81Б Окутин Д. А.

Преподаватель Домрачева А. Б.

Москва 2025

1 Цель работы

Целью работы является формирование общих представлений об анализе и синтезе простых систем, изучение основных понятий в области аналитического моделирования, а также основных понятий теории погрешности.

2 Постановка задачи

Построить модель фрагмента поверхности по заданной матрице высот, обосновать выбор численного метода для приближенного вычисления высоты в заданной точке.

Для реализации цифровой модели использовать триангуляцию Делоне с помощью итеративного алгоритма «Удаляй и строй».

3 Теоретические сведения

3.1 Основные понятия

Триангуляция — планарный граф (без пересечений рёбер), разбивающий плоскость на треугольники.

В триангуляции можно выделить 3 основных вида объектов: узлы (точки, вершины), рёбра (отрезки) и треугольники.

Выпуклая триангуляция — триангуляция, внешняя граница которой образует выпуклый многоугольник (все внутренние углы 180°). Если это условие нарушено, триангуляция считается невыпуклой.

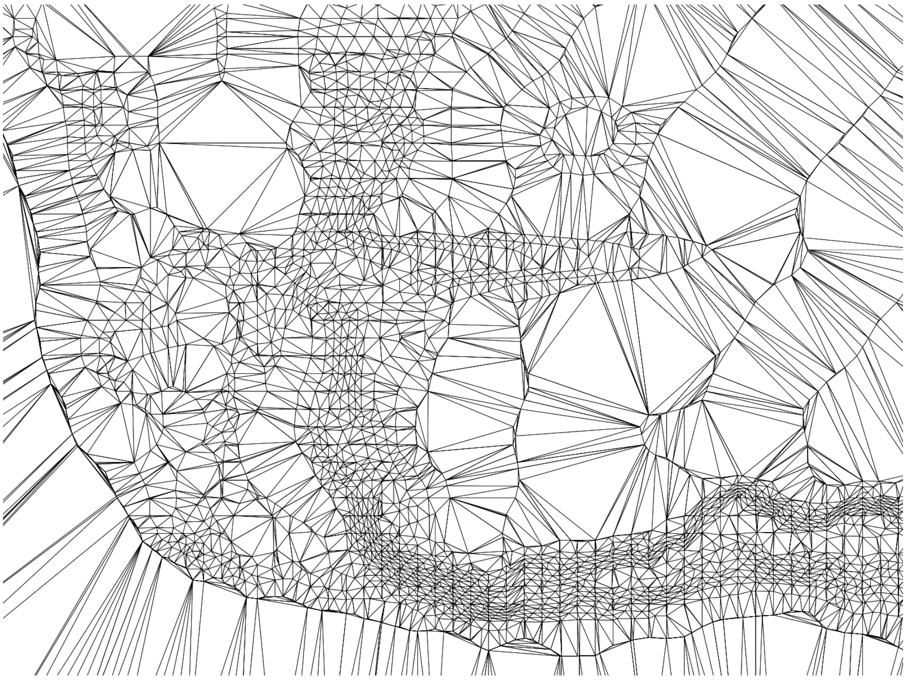


Рис. 1 — Пример триангуляции

Задачей построения триангуляции по заданному набору двумерных точек называется задача соединения заданных точек непересекающимися отрезками так, чтобы образовалась триангуляция. [1]

3.2 Жадная триангуляция

Одним из первых был предложен следующий алгоритм построения триангуляции.

Начало алгоритма.

Шаг 1. Генерируется список всех возможных отрезков, соединяю- щих пары исходных точек, и он сортируется по длинам отрезков.

Шаг 2. Начиная с самого короткого, последовательно выполняется вставка отрезков в триангуляцию. Если отрезок не пересекается с другими ранее вставленными отрезками, то он вставляется, иначе он отбрасывается. Конец алгоритма.

Конец алгоритма.

Если все возможные отрезки имеют разную длину, то результат работы алгоритма однозначен, иначе он зависит от порядка вставки отрезков одинаковой длины. [1]

Жадная триангуляция - триангуляция, построенная жадным алгоритмом.

3.3 Триангуляция Делоне

Условие Делоне — гарантирует, что описанная окружность любого треугольника не содержит других точек множества внутри себя.

Триангуляция Делоне — это выпуклая триангуляция, удовлетворяющая условию Делоне. Она обеспечивает оптимальную форму треугольников, минимизируя «узкие» элементы, что критично для точности алгоритмов интерполяции и визуализации.

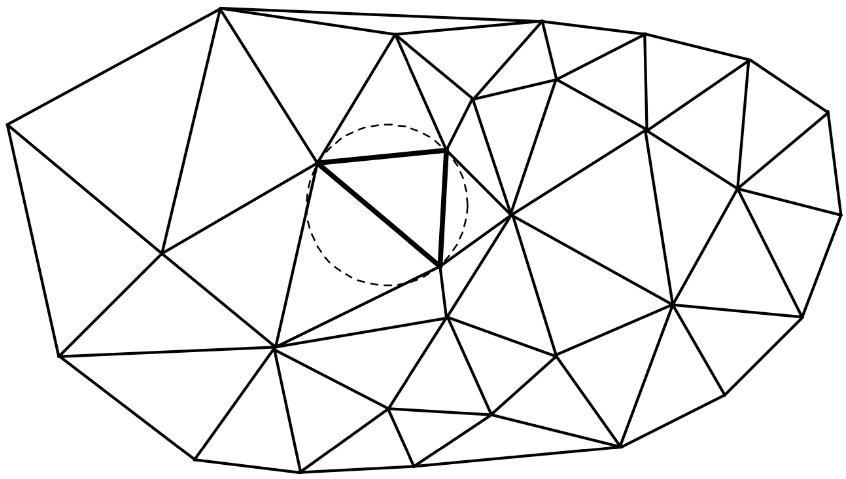
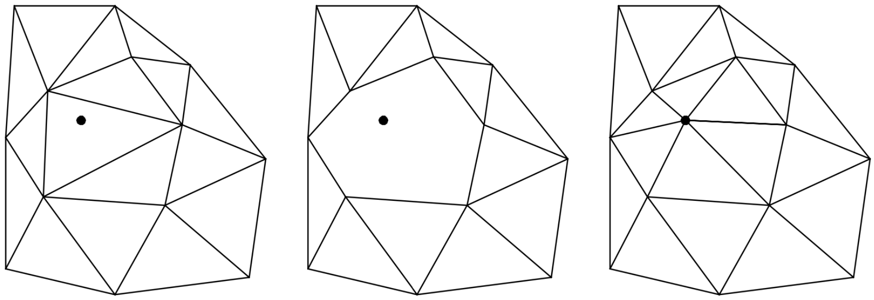


Рис. 2 — Триангуляция Делоне

Существует множество различных алгоритмов реализации триангуляции Делоне, в данной работы был рассмотрен 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 3.



(а) (б) (в)

Рис. 3 — 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.[1]

3.4 Вычисление высоты

Для вычисления высоты в произвольной точке при найденной интерполяции возникает две задачи.

1) Задача определения принадлежности точки треугольнику:

- Вид: Прямая задача (определение положения точки относительно существующих данных).

- Тип: Геометрический поиск (локализация точки в триангуляции).

Необходимо определить, в какой треугольник триангуляции попадает произвольная точка квадрата. Это можно выполнить с помощью метода относительности - необходимо выбрать порядок обхода всех сторон треугольника и для каждого вектора (стороны) понять положение точки относительно неё.

2) Задача интерполяции высоты внутри треугольника:

- Вид: Прямая задача (вычисление значения на основе известных данных).

- Тип: Интерполяция (восстановление значения внутри области по известным значениям в узлах).

Для найденного треугольника, содержащего точку, высота в ней рассчитывается на основе значений высот в вершинах треугольника. Для этого используется линейная интерполяция (барицентрические координаты).

4 Практическая реализация

Далее приведена реализация программы на языке Python, которая генерирует заданное количество точек (координаты x,y,z), находит триангуляцию 2 алгоритмами: жадным алгоритмом и с помощью алгоритма "Удаляй и строй". Затем, используя, найденную триангуляцию происходит вычисление высоты в случайной точке.

Исходный код программы представлен в листинге 1.

Листинг 1: Исходный код

```

import numpy as np

N = 17
np.random.seed(42)

pointsCount = 400
x = np.random.uniform(0, 1000, pointsCount)
y = np.random.uniform(0, 1000, pointsCount)
z = np.random.uniform(0, 100 * N, pointsCount)

points = np.column_stack((x, y, z))

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 8))
plt.scatter(x, y, c=z, cmap='viridis', marker='o')
plt.colorbar(label='                    (   )')
plt.xlabel('X (   )')
plt.ylabel('Y (   )')
plt.title('
                    2D                    ')
plt.grid(True)
plt.show()

import matplotlib.pyplot as plt


def sign(x):
    return np.sign(x)


def cross_product_2d(a, b):
    return a[0] * b[1] - a[1] * b[0]
```

```python


def subtract_vectors(v1, v2):
    return [v1[0] - v2[0], v1[1] - v2[1]]


def are_crossing(v11, v12, v21, v22):
    cut1 = subtract_vectors(v12, v11)
    cut2 = subtract_vectors(v22, v21)

    prod1_1 = cross_product_2d(cut1, subtract_vectors(v21, v11))
    prod1_2 = cross_product_2d(cut1, subtract_vectors(v22, v11))

    if sign(prod1_1) == sign(prod1_2) or prod1_1 == 0 or prod1_2 == 0:
        return False

    prod2_1 = cross_product_2d(cut2, subtract_vectors(v11, v21))
    prod2_2 = cross_product_2d(cut2, subtract_vectors(v12, v21))

    if sign(prod2_1) == sign(prod2_2) or prod2_1 == 0 or prod2_2 == 0:
        return False

    return True

def segment_length(p1, p2):
    return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)

def greedy_triangulation(points):
    n = len(points)
    segments = []

    for i in range(n):
        for j in range(i + 1, n):
            length = segment_length(points[i], points[j])
            segments.append((length, (i, j)))

    segments.sort(key=lambda x: x[0])

    triangulation = []

    for segment in segments:
        i, j = segment[1]
        p1, p2 = points[i], points[j]
        intersects = False

        for existing_segment in triangulation:
```

```
            p3, p4 = points[existing_segment[0]], points[existing_segment[1]]
            if are_crossing(p1, p2, p3, p4):
                intersects = True
                break

        if not intersects:
            triangulation.append((i, j))

    return triangulation

triangulation = greedy_triangulation(points)

#
plt.figure(figsize=(8, 8))
for segment in triangulation:
    p1, p2 = points[segment[0]], points[segment[1]]
    plt.plot([p1[0], p2[0]], [p1[1], p2[1]],color='blue')

plt.plot(points[:, 0], points[:, 1],'o', color='green', label='')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('')
plt.legend()
plt.grid(True)
plt.show()


from collections import defaultdict


def is_point_in_triangle(point, triangle_points):
    def sign(p1, p2, p3):
        return (p1[0] - p3[0]) * (p2[1] - p3[1]) - (p2[0] - p3[0]) * (p1[1] - p3[1])

    d1 = sign(point, triangle_points[0], triangle_points[1])
    d2 = sign(point, triangle_points[1], triangle_points[2])
    d3 = sign(point, triangle_points[2], triangle_points[0])


    has_neg = (d1 < 0) or (d2 < 0) or (d3 < 0)
    has_pos = (d1 > 0) or (d2 > 0) or (d3 > 0)

    return not (has_neg and has_pos)
```

```python


def get_triangles_from_edges(edges, points):
    adjacency_list = defaultdict(set)

    for a, b in edges:
        adjacency_list[a].add(b)
        adjacency_list[b].add(a)

    triangles = set()

    for vertex in adjacency_list:
        for neighbor1 in adjacency_list[vertex]:
            for neighbor2 in adjacency_list[vertex]:
                if neighbor1 < neighbor2 and neighbor2 in adjacency_list[neighbor1]:
                    triangle = tuple(sorted([vertex, neighbor1, neighbor2]))
                    triangles.add(triangle)

    triangle_points_list = [
        [points[i] for i in triangle] for triangle in triangles
    ]

    return triangle_points_list


triangles = get_triangles_from_edges(triangulation, points)


def point_triangle(triangles, point):
    for triangle_points in triangles:
        if is_point_in_triangle(point, triangle_points):
            return triangle_points

    return None


def interpolate_height(point, tri):
    vertices = point_triangle(tri, point)
    if vertices is None:
        raise ValueError("
                              ,
                        .")
    A = vertices[0]
    B = vertices[1]
```

```
    C = vertices[2]

    denom = (B[1] - C[1]) * (A[0] - C[0]) + (C[0] - B[0]) * (A[1] - C[1])
    if denom == 0:
        raise ValueError(" , .")


    w1 = ((B[1] - C[1]) * (point[0] - C[0]) + (C[0] - B[0]) * (point[1] - C[1])) / denom
    w2 = ((C[1] - A[1]) * (point[0] - C[0]) + (A[0] - C[0]) * (point[1] - C[1])) / denom
    w3 = 1 - w1 - w2

    height = w1 * A[2] + w2 * B[2] + w3 * C[2]
    return height

x_test, y_test = 500, 500
target_triangle = point_triangle(triangles,[x_test,y_test])

if not target_triangle is None:
    height = interpolate_height([x_test, y_test], triangles)

    print(f" ({x_test}, {y_test}): {height:.2f} ")

    plt.figure(figsize=(8, 8))
    for segment in triangulation:
        p1, p2 = points[segment[0]], points[segment[1]]
        plt.plot([p1[0], p2[0]], [p1[1], p2[1]],color='black', alpha=0.25)

    # plt.plot(points[:, 0], points[:, 1],'o', color='green', label=' ')
    plt.scatter(x, y, c=z, cmap='viridis', marker='o')
    plt.colorbar(label=' ( )')

    p1,p2,p3 = target_triangle
    plt.plot([p1[0], p2[0]], [p1[1], p2[1]],color='purple')
    plt.plot([p1[0], p3[0]], [p1[1], p3[1]],color='purple')
    plt.plot([p3[0], p2[0]], [p3[1], p2[1]],color='purple')
    plt.plot(x_test, y_test,'o', color='purple', label=' ')

    plt.xlabel('X')
```

```
    plt.ylabel('Y')
    plt.title('                                              ')
    plt.legend()
    plt.grid(True)
    plt.show()
else:
    print(f"              ({x_test}, {y_test})                                      ")


def circumcircle(A, B, C, tol=1e-12):
    d = 2 * (A[0]*(B[1]-C[1]) + B[0]*(C[1]-A[1]) + C[0]*(A[1]-B[1]))
    if abs(d) < tol:
        return None, None
    A_sq = A[0]**2 + A[1]**2
    B_sq = B[0]**2 + B[1]**2
    C_sq = C[0]**2 + C[1]**2
    center_x = (A_sq*(B[1]-C[1]) + B_sq*(C[1]-A[1]) + C_sq*(A[1]-B[1])) / d
    center_y = (A_sq*(C[0]-B[0]) + B_sq*(A[0]-C[0]) + C_sq*(B[0]-A[0])) / d
    center = np.array([center_x, center_y])
    r_sq = np.sum((A - center)**2)
    return center, r_sq

def delete_and_build_triangulation(points, tol=1e-12):
    n = len(points)

    xmin, xmax = np.min(points[:, 0]), np.max(points[:, 0])
    ymin, ymax = np.min(points[:, 1]), np.max(points[:, 1])
    dx = xmax - xmin
    dy = ymax - ymin
    delta = max(dx, dy)
    midx = (xmin + xmax) / 2
    midy = (ymin + ymax) / 2
    A = np.array([midx - 2*delta, midy - delta])
    B = np.array([midx, midy + 2*delta])
    C = np.array([midx + 2*delta, midy - delta])
    supertriangle = np.array([A, B, C])

    points_ext = np.vstack([points, supertriangle])
    triangulation = [(n, n+1, n+2)]

    for i, p in enumerate(points):
        bad_triangles = []
        for tri in triangulation:
```

```
            A_idx, B_idx, C_idx = tri
            A_pt = points_ext[A_idx]
            B_pt = points_ext[B_idx]
            C_pt = points_ext[C_idx]
            center, r_sq = circumcircle(A_pt, B_pt, C_pt, tol)
            if center is None:
                continue
            if np.sum((p - center)**2) < r_sq - tol:
                bad_triangles.append(tri)

        edge_count = {}
        for tri in bad_triangles:
            edges = [(tri[0], tri[1]), (tri[1], tri[2]), (tri[2], tri[0])]
            for edge in edges:
                edge_sorted = tuple(sorted(edge))
                edge_count[edge_sorted] = edge_count.get(edge_sorted, 0) + 1

        boundary_edges = [edge for edge, count in edge_count.items() if count == 1]

        triangulation = [tri for tri in triangulation if tri not in bad_triangles]

        for edge in boundary_edges:
            new_tri = (edge[0], edge[1], i)
            triangulation.append(new_tri)

    final_triangles = [tri for tri in triangulation if all(v < n for v in tri)]

    return final_triangles

triangles_dealoune = delete_and_build_triangulation(points[:, :2])

def convert_to_point_trinagles(triangles):
    ans = []
    for tri in triangles:
        p1,p2,p3 = tri
        ans.append([points[p1],points[p2],points[p3]])

    return ans

triangles_dealoune_points = convert_to_point_trinagles(triangles_dealoune)
```

```
x_test, y_test = 500, 500
target_triangle = point_triangle(triangles_dealoune_points,[x_test,
    y_test])

if not target_triangle is None:
     height = interpolate_height([x_test, y_test],
    triangles_dealoune_points)

     print(f"                              ({x_test}, {y_test}): {height:.2f
    }    ")

     plt.figure(figsize=(8, 8))
     for tri in triangles_dealoune:
          pts = points[list(tri)]
          pts = np.vstack([pts, pts[0]])   #

          plt.plot(pts[:, 0], pts[:, 1], color="black", alpha=0.25)

    # plt.plot(points[:, 0], points[:, 1],'o', color='green', label='
                ')
     plt.scatter(x, y, c=z, cmap='viridis', marker='o')
     plt.colorbar(label='                  (    )')

     p1,p2,p3 = target_triangle
     plt.plot([p1[0], p2[0]], [p1[1], p2[1]],color='purple')
     plt.plot([p1[0], p3[0]], [p1[1], p3[1]],color='purple')
     plt.plot([p3[0], p2[0]], [p3[1], p2[1]],color='purple')
     plt.plot(x_test, y_test,'o', color='purple', label='
                 ')

     plt.xlabel('X')
     plt.ylabel('Y')
     plt.title('                                                ')
     plt.legend()
     plt.grid(True)
     plt.show()
else:
     print(f"                ({x_test}, {y_test})
                                         ")
```

5 Результаты

Результаты визуализаций траектории представлены на рисунках 4 - 5.

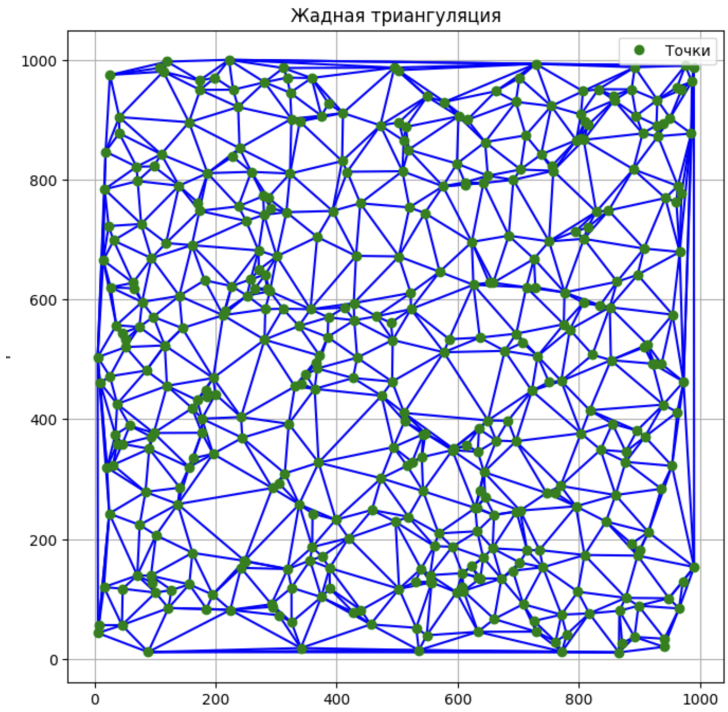


Рис. 4 — Триангуляция жадным алгоритмом

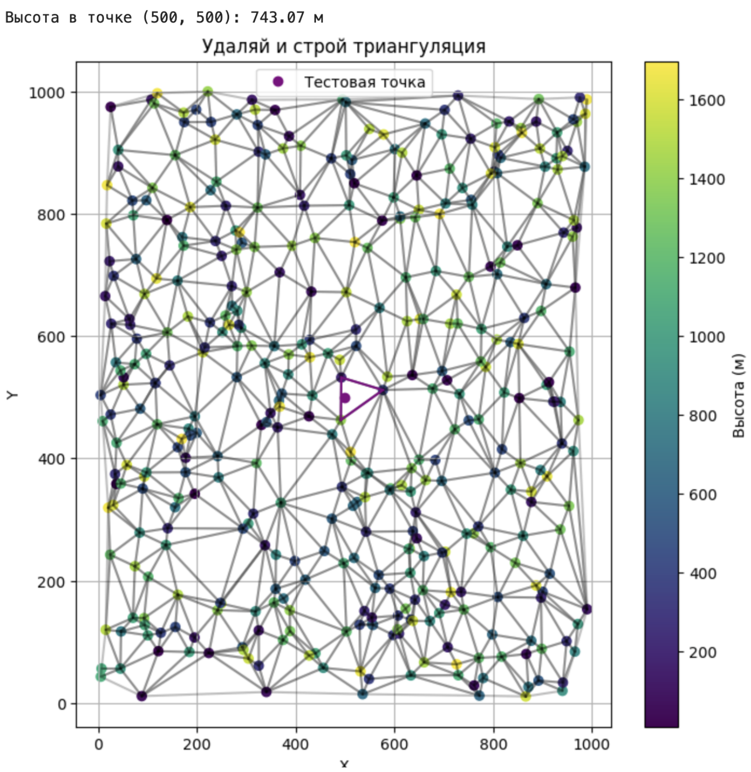


Рис. 5 — Триангуляция алгоритмом "Удаляй и строй"
с нахождением высоты в точке

6 Заключение

В ходе выполнения лабораторной работы был изучен метод триангуляции Делоне, реализованы: жадный алгоритм триангуляции и итеративный алгоритм «Удаляй и строй» на языке программирования Python. Также было реализовано нахождение высоты в произвольной точки внутри квадрата.

В ходе работы было выявлено, что жадный алгоритм интерполяции работает значительно дольше, чем алгоритм триангуляции Делоне (алгоритм "Удаляй и строй"), при это оба алгоритма строят визуально схожую триангуляцию.

Функция подсчета высоты в точке функционирует исправно и возвращает корректные значения. Однако данное значение невозможно считать достаточно точным, т.к оно высчитывается в зависимости от близлижайших точек треугольника, внутрь которого попадает целевая точка.

В данной лабораторной работе триангуляция рассматривалась на плоскости и в том числе с помощью неё выполнялась интерполяция высоты в точке, однако в дальнейшем можно применить метод триангуляции Делоне для моделирования рельефа местности, подключив третью координату.

7 Список литературы

1) А.В. Скворцов. Триангуляция Делоне и её применение. – Томск: Изд-во Том. ун-та, 2002. – 128 с.