Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №2

«Триангуляция Делоне»

по курсу «Моделирование»

Выполнила:

студентка 4 курса,

группы ИУ9-82

Козлова А. А.

Проверила:

Домрачева А. Б.

2018 г

Постановка задачи:

Изучить различные алгоритмы триангуляции Делоне и реализовать простой итеративный алгоритм.

1. Теоретические сведения.

Триангуляция (от лат. triangulum – треугольник) – планарное разбиение плоскости на N фигур, из которых одна является внешней неограниченной, а остальные – треугольниками.

Задачей построения триангуляции по заданному на- бору двумерных точек называется задача соединения заданных точек не- пересекающимися отрезками так, чтобы образовалась триангуляция

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.

Триангуляция называется триангуляцией Делоне, если она является выпуклой и удовлетворяет условию Делоне [1].

1.2 Итеративные алгоритмы построения триангуляции Делоне

Все итеративные алгоритмы триангуляции Делоне основываются на последовательном добавлении точек в частично построенную триангуляцию. Пусть имеется триангуляция Делоне из $n-1$ точки, тогда при добавлении очередной n-й точки надо выполнить следующие шаги.

Итеративный алгоритм построения триангуляции Делоне:

1. Локализовать точку, т.е. найти построенный ранее треугольник, в который попадает точка. Если точка попадает не внутрь триангуляции, то найти ближайший к ней треугольник.

2. Делаем один из следующих шагов в зависимости от положения точки.

2.1. Если точка попала на ранее вставленную, то она, как правило, отбрасывается.

2.2. Если точка попала на ребро, то оно разбивается на два новых. Оба смежных треугольника также делятся на два меньших.

2.3. Если точка попала строго внутрь какого-нибудь треугольника, то он делится на три новых.

2.4. Если точка попала вне триангуляции, то строится один или более новых.

3. После добавления новой точки условие Делоне может быть нарушено, поэтому надо проверить все вновь построенные треугольники и соседние с ними.

1.2. Простой итеративный алгоритм

В простом итеративном алгоритме поиск очередного треугольника реализуется следующим образом. Берётся любой треугольник, уже принадлежащий триангуляции (например, выбирается случайно), и последовательными переходами по связанным треугольникам ищется искомый треугольник.

При этом в худшем случае приходится пересекать все треугольники триангуляции, поэтому трудоемкость такого поиска составляет O(N).

Во многих практически важных случаях исходные точки не являются статистически независимыми, при этом точка i находится вблизи точки i+1. Поэтому в качестве начального треугольника для поиска можно брать треугольник, найденный ранее для предыдущей точки. Тем самым иногда удается достичь на некоторых видах исходных данных трудоемкости построения триангуляции в среднем O(N).

Для правильной работы данного алгоритма поиска существенным является то, что в триангуляции выполняется условие Делоне. Если условие Делоне нарушено, то иногда возможно зацикливание алгоритма.

После того как требуемый треугольник найден, в нем строятся новые узел, рёбра и треугольники, а затем производится локальное перестроение триангуляции.

2. Реализация

2.1 Выбор суперструктуры.

Для упрощения алгоритма, можно исключить случай, когда новая точка попадает в область вне триангуляции. С этой целью изначально в триангуляцию были добавлены несколько дополнительных узлов таких, что построенная на них триангуляция заведомо накрывает все исходные точки. Такая структура обычно называется суперструктурой.

В качестве суперструктуры был выбран квадрат. Поэтому вначале в триангуляцию добавляются два треугольника, полученные путем проведения одной из диагоналей квадрата.

```
pointsInTriangulation = [
 Point(500., 500.),
   Point(500., -500.),
   Point(-500., -500.),
   Point(-500., 500.),
]
triangles = [Triangles(pointsInTriangulation[0],
              pointsInTriangulation[1],
              pointsInTriangulation[3],
               ),
         Triangles(pointsInTriangulation[2],
               pointsInTriangulation[3],
               pointsInTriangulation[1],
               ),
         ]
 # добавляем друг друга в соседи
 triangles[0].addNewNeighborT(triangles[1])
```

Листинг 1. Создание суперструктуры.

2.2. Выбор способа поиска треугольника, в который добавляется точка.

Для поиска треугольника по заданной точке внутри него и по некоторому исходному треугольнику был выбран следующий способ: двигаться пошагово, на каждом из которых надо переходить через такое ребро текущего треугольника, что целевая точка и вершина текущего

треугольника, противолежащая выбираемому пересекаемому ребру, лежат по разные стороны от прямой, определяемой данным ребром (рис. 2).

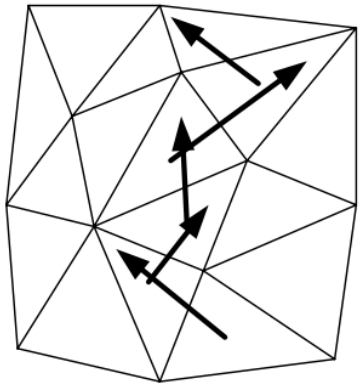


Рис 2. Вариант локализации треугольника в итеративных алгоритмах: переход через разделяющее ребро

Этот способ обычно обеспечивает более длинный путь до цели, но он алгоритмически проще и поэтому быстрее [1].

```
# обработка оставшихся точек
for id_point in range(0, len(points)):
    cur = triangulation(id_point, triangles[cur])
    pointsInTriangulation.append(points[id_point])
```

Листинг 2. Обработка точек триангуляции, не принадлежащих суперструктуре.

```
def triangulation(id_point, triangle):
global triangles, points, pointsInTriangulation, shouldDeleted
 print("triangulation")
 cur = triangles.index(triangle)
 # проверка на совпадение точек
 if points[id_point] not in pointsInTriangulation:
    # проверяем принадлежит ли точка ребру
    for id_edge in range(3):
       if (triangles[cur].edge[id_edge].onEdge(points[id_point])):  # если принадлежит
           addTriangles = triangulationIfBelongsEdge(id_point, cur, id_edge)
           rebuild(addTriangles)
           return cur
```

```
13.       # проверяем с одной ли строны точка и противоположная вершина от ребра текущего
треугольника
14.       # если с другой стороны стороны
15.        elif (not triangles[cur].edge[id_edge].isOneSide1(points[id_point])):
16.           triangle = triangles[cur].neighbors.get(id_edge)
17.           cur = triangles.index(triangle)
18.           triangulation(id_point, triangle)
19.           return cur
20.       # значит точка внутри текущего треугольника
21.       # возвращаем (id, номер общего ребра)
22.       addTriangles = insideTriangle(id_point, triangle)
23.       rebuild(addTriangles)
24.       return cur
```

Листинг 3. Функция Triangulation, в которой осуществляется поиск треугольника, в который добавляется точка.

```
1. def insideTriangle(k, triangle):
2.   global triangles, shouldDeleted
3.   neighbords = []
4.   newTriangles = []
5.   id_1 = len(triangles)
6.   cur = triangles.index(triangle)
7.   for i in range(0, 3):
8.      triangles.append(Triangles(
9.         triangles[cur].edge[i].v1,
10.         points[k],
11.         triangles[cur].edge[i].v2, )
12.      )
13.      if (triangles[cur].neighbors.get(i) != None):
14.         # находим id соседа по ребру
15.         neighbordId = triangles.index(triangles[cur].neighbors.get(i))
16.         (idNewTedge, idOldTedge) = triangles[id_1 +
i].addAlreadyExistNeighborT(triangles[neighbordId])
17.         # переделать в commonEdge = {T : edge,...} и удалять старый T в конце
18.         newTriangles.append([id_1 + i, idNewTedge])
19.         neighbords.append([neighbordId, idOldTedge])
```

```
    # добавляем друг с другом
    for i in range(id_1 + 1, id_1 + 3):
        triangles[id_1].addNewNeighborT(triangles[i])
    triangles[id_1 + 1].addNewNeighborT(triangles[id_1 + 2])
    triangles.pop(cur)
    return [triangles[len(triangles) - i] for i in range(1, 4)]
```

Листинг 4. Функция insideTriangle, в которой обрабатывается вариант, когда новая точка попадает во внутрь треугольника.

```
def triangulationIfBelongsEdge(k, cur, idEdgeCur):
    print("belong")
    global triangles, points
    id_1 = len(triangles)
    triangles.append(Triangles(triangles[cur].edge[idEdgeCur].v1,
                               points[k],
                               triangles[cur].edge[idEdgeCur].oppositePoint,
                               )
                     )
    triangles.append(Triangles(triangles[cur].edge[idEdgeCur].v2,
                               points[k],
                               triangles[cur].edge[idEdgeCur].oppositePoint,
                               ))
    # находим id соседа по ребру, на которое попадает точка
    neighbordId = triangles.index(triangles[cur].neighbors.get(idEdgeCur))
    # вычисляем id того же ребра в соседнем треугольнике
    idEdgeNeighb = triangles[cur].findAnalogEdge(triangles[neighbordId], idEdgeCur)
    triangles.append(Triangles(triangles[neighbordId].edge[idEdgeNeighb].v1,
                               points[k],
                               triangles[neighbordId].edge[idEdgeNeighb].oppositePoint,
                               )
                     )
    triangles.append(Triangles(triangles[neighbordId].edge[idEdgeNeighb].v2,
                               points[k],
                               triangles[neighbordId].edge[idEdgeNeighb].oppositePoint,
                               )
```

```
            )
# добавить соседей только что созданных соседей
for i in range(id_1, len(triangles)):
    for id in range(i + 1, len(triangles)):
        add = triangles[id].addNewNeighborT(triangles[i])
setCurEdges = [0, 1, 2]
setNeighbordEdges = [0, 1, 2]
setCurEdges.pop(idEdgeCur)
setNeighbordEdges.pop(idEdgeNeighb)
# добавить уже существующих соседей
# надо получить id ребра в соседнем треугольнике
for i in setCurEdges:
    if (triangles[cur].neighbors.get(i) != None):
        id_neighb_cur = triangles.index(triangles[cur].neighbors.get(i))
        for id in range(id_1, id_1 + 2):
            triangles[id].addAlreadyExistNeighborT(triangles[id_neighb_cur])
trianglesN = triangles[neighbordId]

for i in setNeighbordEdges:
    if (triangles[neighbordId].neighbors.get(i) != None):
        id_neighb_neighb = triangles.index(triangles[neighbordId].neighbors.get(i))
        print(id_neighb_neighb)
        for id in range(id_1 + 2, len(triangles)):
            triangles[id].addAlreadyExistNeighborT(triangles[id_neighb_neighb])
triangles.pop(cur)
triangles.pop(triangles.index(trianglesN))
return [triangles[len(triangles) - i] for i in range(1, 5)]
```

Листинг 5. Функция triangulationIgBelongsEdge, в которой обрабатывается вариант, когда новая точка попадает на ребро треугольника.

```
def rebuild(addTriangles):
global triangles, shouldDeleted
for triangle in addTriangles:
    id_addT = triangles.index(triangle)
    # находим соседа по каждому из ребер треугольников, добавленных на предыдущем шаге
    for j in range(3):
```

```
        neighbor = triangles[id_addT].neighbors.get(j)
        if (neighbor != None and neighbor not in addTriangles):
          id_neighbor = triangles.index(neighbor)
          id_edgeAddT,id_edgeOldT = triangles[id_addT].findAnalogEdge(triangles[id_neighbor])
          oppositePointAddT = triangles[id_addT].edge[id_edgeAddT].oppositePoint
          oppositePointOldT = triangles[id_neighbor].edge[id_edgeOldT].oppositePoint
          setAddTEdges = [0, 1, 2]
          setOldTEdges = [0, 1, 2]
          setAddTEdges.pop(id_edgeAddT)
          setOldTEdges.pop(id_edgeOldT)
           # проверка на условие Делане и выпуклость четырехугольника
          if (not condDelaune2(triangles[id_neighbor], oppositePointAddT) and
               convex(id_addT, id_neighbor, setAddTEdges, setOldTEdges)):
              shouldDeleted.append(triangles[id_addT])
              shouldDeleted.append(triangles[id_neighbor])
              len_ = len(triangles)
              triangles.append(Triangles(oppositePointAddT,
                              oppositePointOldT,
                              triangles[id_addT].edge[id_edgeAddT].v1)
                          )
              triangles.append(Triangles(oppositePointAddT,
                              oppositePointOldT,
                              triangles[id_addT].edge[id_edgeAddT].v2)
                          )
              # добавляем в соседи друг друга
              triangles[len_].addNewNeighborT(triangles[len_ + 1])
              #
              for k in setOldTEdges:
                if triangles[id_neighbor].neighbors.get(k) != None:
                  id_neighb = triangles.index(triangles[id_neighbor].neighbors.get(k))
                  for id in range(len_, len_ + 2):
                    triangles[id].addAlreadyExistNeighborT(triangles[id_neighb])

              for j in setAddTEdges:
                if triangles[id_addT].neighbors.get(j) != None:
                  id_neighb = triangles.index(triangles[id_addT].neighbors.get(j))
                  for id in range(len_, len_ + 2):
                    triangles[id].addAlreadyExistNeighborT(triangles[id_neighb])

              for id in range(len(shouldDeleted)):
                triangles.pop(triangles.index(shouldDeleted[id]))
              shouldDeleted.clear()
              break
    return
```

Листинг 6. Функция rebuild, в которой проверяется выполнение условия Делоне для созданных треугольников и выполняются изменения, если условие не выполняется.

2. 3. Способ проверки условия Делане.

Для того, чтобы посмотреть выполняется ли условие Делане, был выбран способ проверки с заранее вычисленной описанной окружностью.

Основная идея алгоритма проверки через заранее вычисленные окружности заключается в предварительном вычислении для каждого построенного треугольника центра и радиуса описанной вокруг него окружности, после чего проверка условия Делоне будет сводиться к вычислению расстояния до центра этой окружности и сравнению результата с радиусом.

Уравнение окружности, проходящей через точки $(x_1, y_1),(x_2, y_2),(x_3, y_3)$ можно записать в виде:

$$\begin{vmatrix} x^2+y^2 & x & y & 1 \\ x_1^2+y_1^2 & x_1 & y_1 & 1 \\ x_2^2+y_2^2 & x_2 & y_2 & 1 \\ x_3^2+y_3^2 & x_3 & y_3 & 1 \end{vmatrix} \quad \begin{vmatrix} x_1^2+y_1^2 & x_1 & y_1 & 1 \\ x_2^2+y_2^2 & x_2 & y_2 & 1 \\ x_3^2+y_3^2 & x_3 & y_3 & 1 \end{vmatrix}$$

или $(x^2+y^2)*a-x*b+y*c-d=0$, где

$$a=\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} , \quad b=\begin{vmatrix} x_1^2+y_1^2 & y_1 & 1 \\ x_2^2+y_2^2 & y_2 & 1 \\ x_3^2+y_3^2 & y_3 & 1 \end{vmatrix} ,$$

$$c=\begin{vmatrix} x_1^2+y_1^2 & x_1 & 1 \\ x_2^2+y_2^2 & x_2 & 1 \\ x_3^2+y_3^2 & x_3 & 1 \end{vmatrix} , \quad d=\begin{vmatrix} x_1^2+y_1^2 & x_1 & y_1 \\ x_2^2+y_2^2 & x_2 & y_2 \\ x_3^2+y_3^2 & x_3 & y_3 \end{vmatrix}$$

Тогда условие Делоне для любого заданного треугольника Δ $(x_1, y_1),(x_2, y_2),(x_3, y_3)$ будет выполняться только тогда, когда для любого узла (x_0, y_0) триангуляции будет $((x^2+y^2)*a-x*b+y*c-d)*sign\,a\geq 0$, т.е. когда

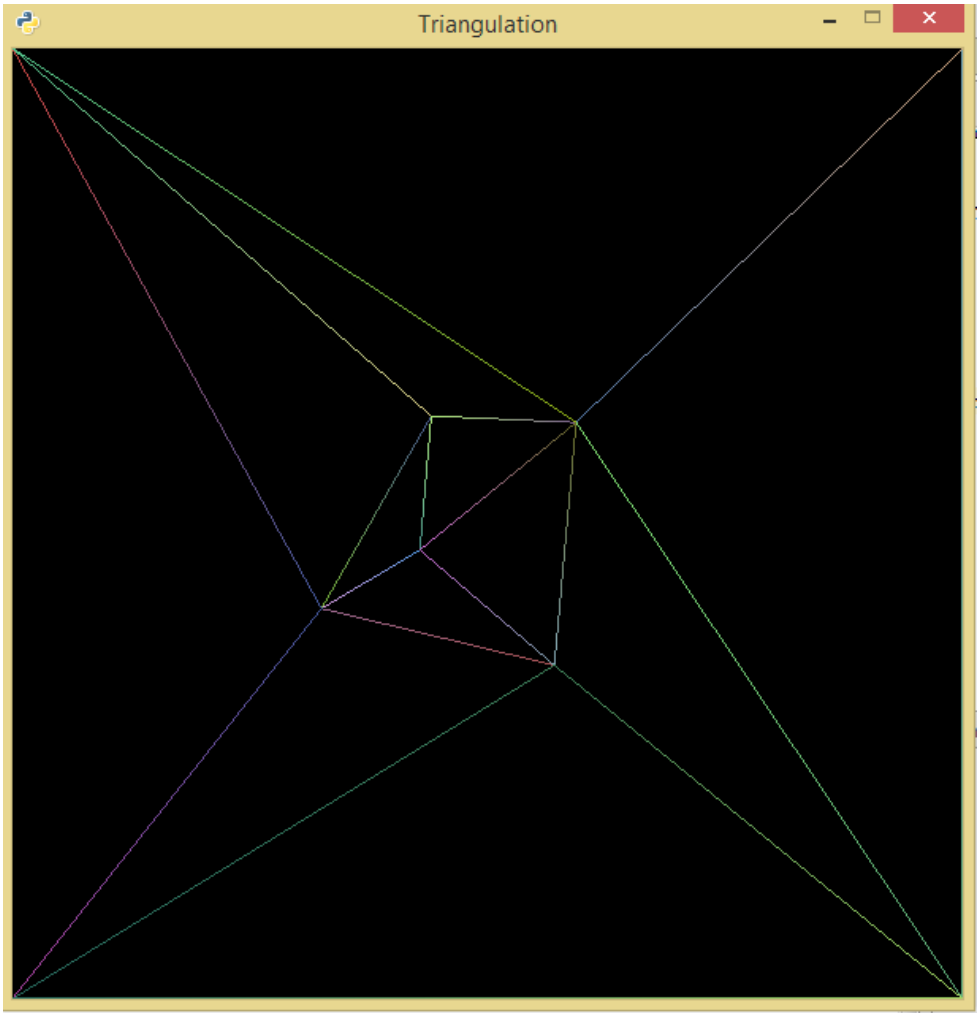
(x_0, y_0) не попадает внутрь окружности, описанной вокруг треугольника Δ $(x_1, y_1), (x_2, y_2), (x_3, y_3)$.

```
1.  def condDelaune(triangle, point):
2.  a = np.array([[triangle.v[i].x, triangle.v[i].y, 1] for i in range(len(triangle.v))])
3.  a = linalg.det(a)
4.  b = np.array(
5.    [[pow(triangle.v[i].x, 2) + pow(triangle.v[i].y, 2), triangle.v[i].y, 1] for i in
6.     range(len(triangle.v))])
7.  b = linalg.det(b)
8.  c = np.array(
9.    [[pow(triangle.v[i].x, 2) + pow(triangle.v[i].y, 2), triangle.v[i].x, 1] for i in
10.    range(len(triangle.v))])
11.  c = linalg.det(c)
12.  d = np.array(
13.[[pow(triangle.v[i].x, 2) + pow(triangle.v[i].y, 2), triangle.v[i].x, triangle.v[i].y]
14.for i in  range(len(triangle.v))])
15.  d = linalg.det(d)
16.  x = b / (2 * a)
17.  y = -c / (2 * a)
18.  center = Point(x, y)
19.  for i in range(3):
20.    sign = a / abs(a)
21.    res = (a * (point.x ** 2 + point.y ** 2) - b * triangle.v[i].x + c * triangle.v[i].y -
d) * sign
22.    if (res < 0):
23.      return False
24.  return True
```

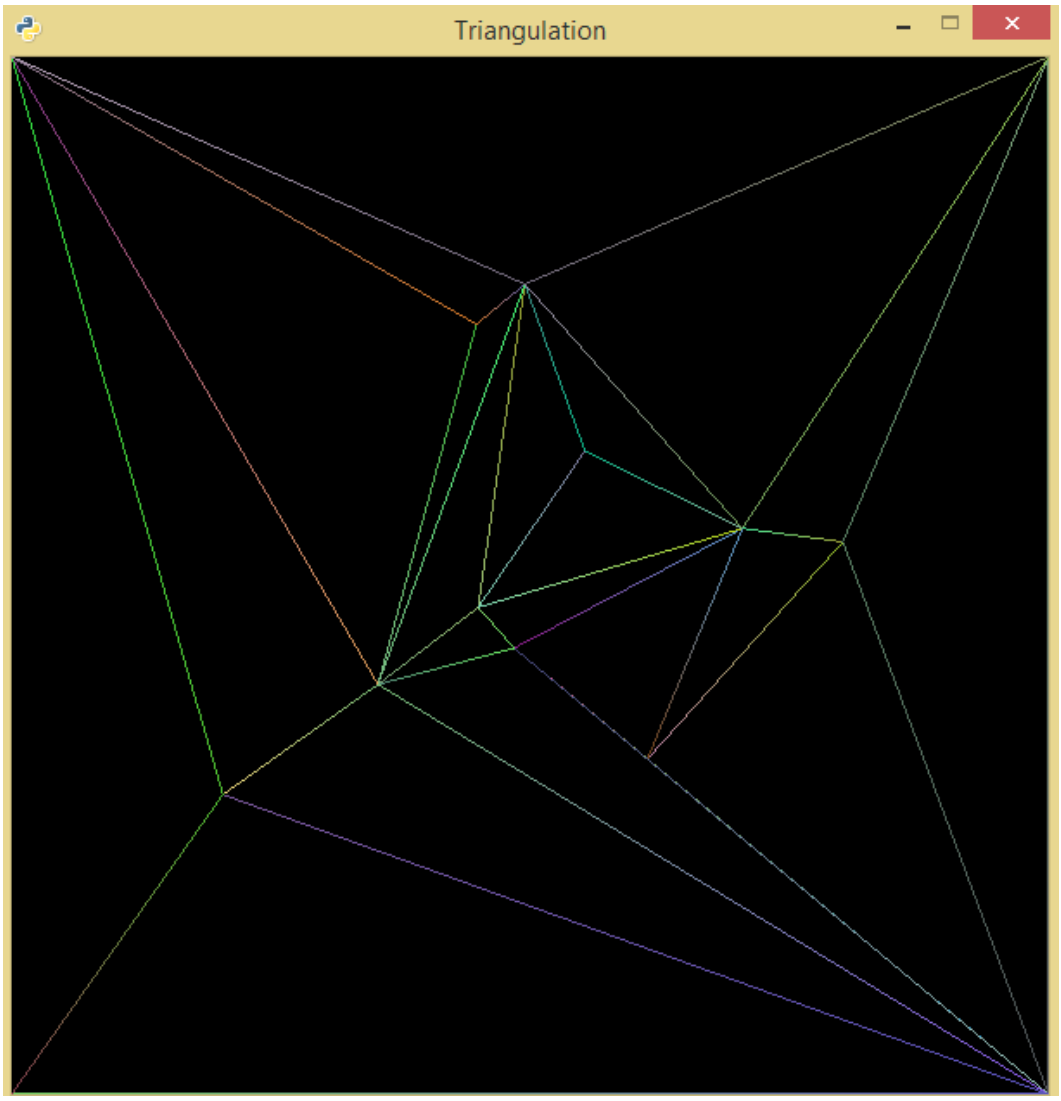
Листинг 6. Функция conDelone, в которая реализует проверку условия Делоне.

Тестирование.

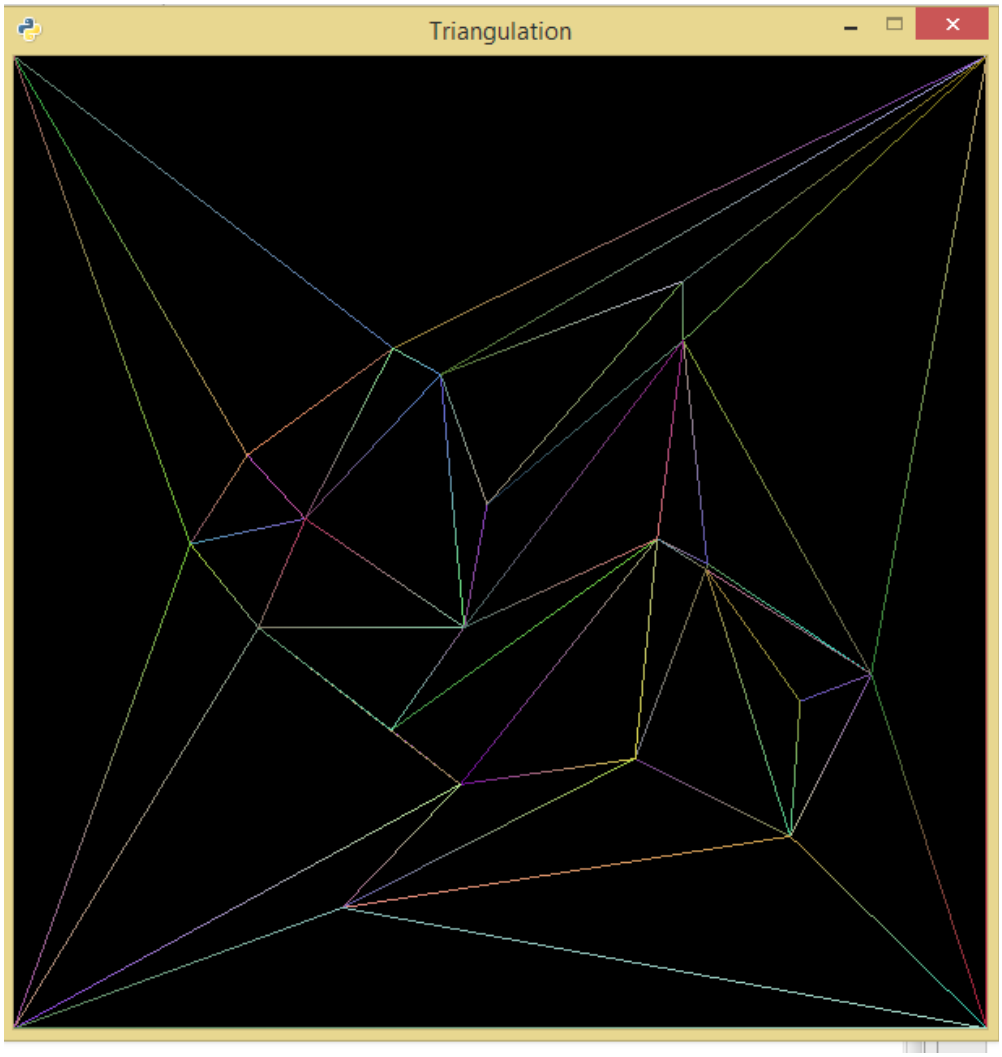
Результат работы программы для 5 точек:



Результат работы программы для 10 точек:



Результат работы программы для 20 точек:



Вывод:

В ходе выполнения лабораторной работы было изучены алгоритмы построения триангуляции Делоне и реализован простой итеративный алгоритм, который работает быстрее неоптимизированного алгоритма триангуляции засчет того, что в качестве начального треугольника для поиска берется треугольник, найденный ранее для предыдущей точки

Использованная литература:

1. А.В. Скворцов «Триангуляция Делоне и её применение »