Аннотация

При осуществлении топографической съемки пещер проводятся измерения, с помощью которых, после окончания процесса съемки, строится трёхмерная модель хода пещеры и находятся ее морфометрические характеристики (площадь, протяженность, объем, амплитуда и т.д.). Если обрабатывать данные вручную, то это занимает большое количество времени, также увеличиваются риски возникновения ошибок и неточностей, вызванных человеческим фактором. Всё вышеперечисленное показывает необходимость программного решения данной задачи.

Настоящая дипломная работа посвящена разработке и реализации алгоритмов, вычисления площади и объёма трёхмерной модели пещеры по пикетам и отстрелам, полученным в ходе топографической съёмки. Для этого сначала рассчитаем координаты пикетов и отстрелов в трёхмерном пространстве. Затем мы последовательно ищем площадь и объём участков пещеры, работая с парами соседних пикетов. На финальном этапе мы суммируем найденные значения, и получаем площадь и объём всей пещеры.

В заключении описываются результаты, предоставляемые реализацией каждого алгоритма, проводится сравнение эффективности алгоритмов, а также приводятся соображения о возможных путях дальнейшей работы над задачей.

Оглавление

Введение

Дабы определить размер и структуру пещеры, проводится ряд мероприятий, таких как спелеосъемка или топографическая съемка пещеры. Процесс съемки реализован следующим образом: несколько человек (обычно не более трех), расставляют пикеты от входа в пещеру и следуя далее по ее ходу; производят измерения азимута между ними, расстояния и угла наклона. После проведения топографической съемки строится топографический план пещеры, который затем используется для решения различных задач.

При помощи таких компьютерных программ как: Walls, Karst, Therion, Compass, Топо возможно построение топосъемок. Данные программные средства позволяют получить наглядное представление координат точек в пространстве. В данной работе была использована программа Топо.

Чтобы найти площадь и объем хода трехмерной модели пещеры, нужно использовать расположение пикетов и соответствующих им отстрелов.

В настоящей работе решается задача разработки алгоритмов поиска площади и объема частей хода пещеры и написания соответствующего программного кода на языке C++. Входными данными служит массив пикетов и отстрелов.

Глава 1. Исследование предметной области

1.1 Основные понятия

Пещера — это естественная полость в верхней толще земной коры, сообщающаяся с её поверхностью одним или несколькими отверстиями. Существует множество различных типов пещер (лавовые, антропогенные, прибрежные, ледовые и другие). Пещеры изучает наука спелеология.

Спелеология — это наука о пещерах, их структуре, физических свойствах, истории, формах жизни и процессах, в которых они образуются (спелеогенез) и изменяются во времени (спелеоморфология). Спелеология — это междисциплинарная область, объединяющая знания химии, биологии, геологии, физики, метеорологии и картографии для создания портретов пещер как сложных, развивающихся систем.

Спелеотопосъемка — это комплекс топографо-геодезических работ для определения формы и размеров пещеры. Смысл топографической съемки (топосъемки) заключается в том, чтобы определить взаимное расположение некоторого множества точек относительно выбранной системы отсчета или друг друга. Топосъемка пещеры происходит следующим образом: расставляют пикеты, измеряют между ними расстояние, замеряют азимут и вертикальный угол. Все замеры записывают в специальный журнал, который называется пикетажным журналом. По этим записям впоследствии выстраивают топографическую карту.

Топографическая карта — общегеографическая карта универсального назначения, подробно изображающая местность. Такие карты используются, чтобы показать рельеф и особенности области для архитекторов, инженеров истроительных подрядчиков. При составлении карт пещеры задача состоит

в том, чтобы получить проекцию ходов пещеры на горизонтальную плоскость.

Пикет – это точка, в которой производятся измерения, а путь, или нитка – набор соединенных пикетов. Определение расстояния до стен, пола и потолка необходимы для нахождения объема хода.

Отстрел пикета (сокр. отстрел) – вектор от пикета до некоторой точки стены.

Азимут — это угол между направлением на магнитный север и направлением на предмет, причем этот угол отсчитывается по часовой стрелке и может иметь значения от 0 до 360 градусов.

1.2 Средства реализации

Для представления трехмерных моделей пещер и вычисления примерных значения объёма и площади мы будем использовать программу Топо. Данная программа позволяет строить нитку хода, разрез-развертку, границы пещеры, сечения поверхности, визуализировать рельеф и делать привязку пещер к рельефу по географическим координатам и многое другое.

С помощью Топо рассматривалось положение пикетов и отстрелов в пространстве.

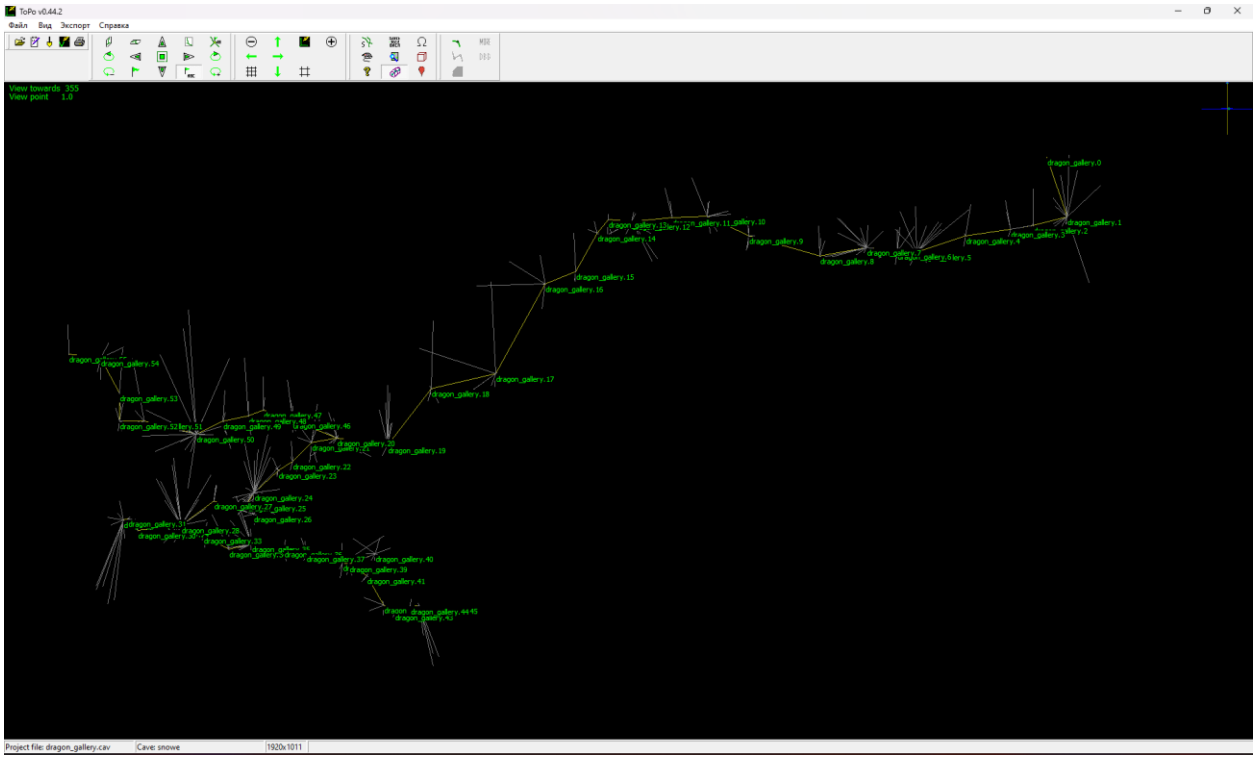


Рис 1.2. 1. Изображение точек пещеры «Галерея дракона»

Но чаще всего программа использовалась для построения планов и вертикальных проекций.

С помощью планов были определены примерные площади участок пещер. А вертикальные проекции позволяли найти среднюю высоту определенного участка пещеры.

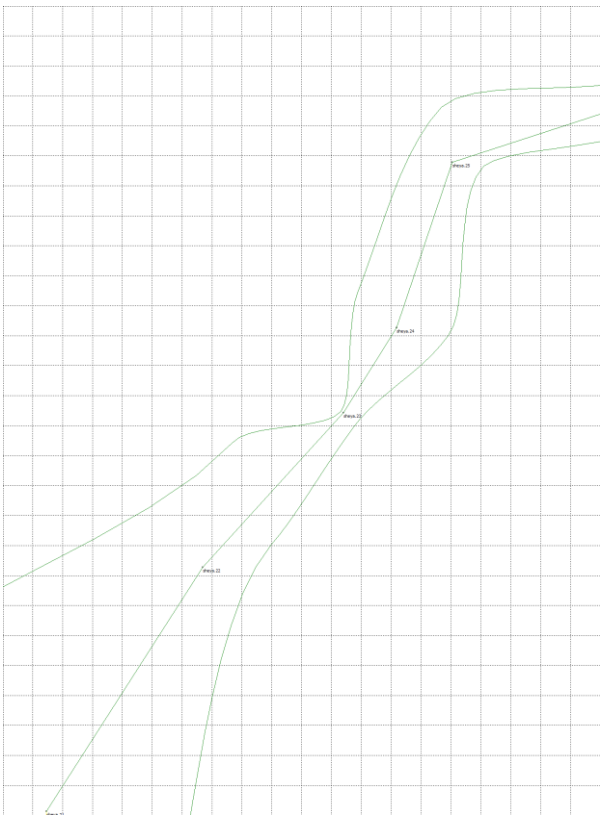


Рис 1.2.2. План, показывающий пространство между несколькими пикетам пещеры «Шея»

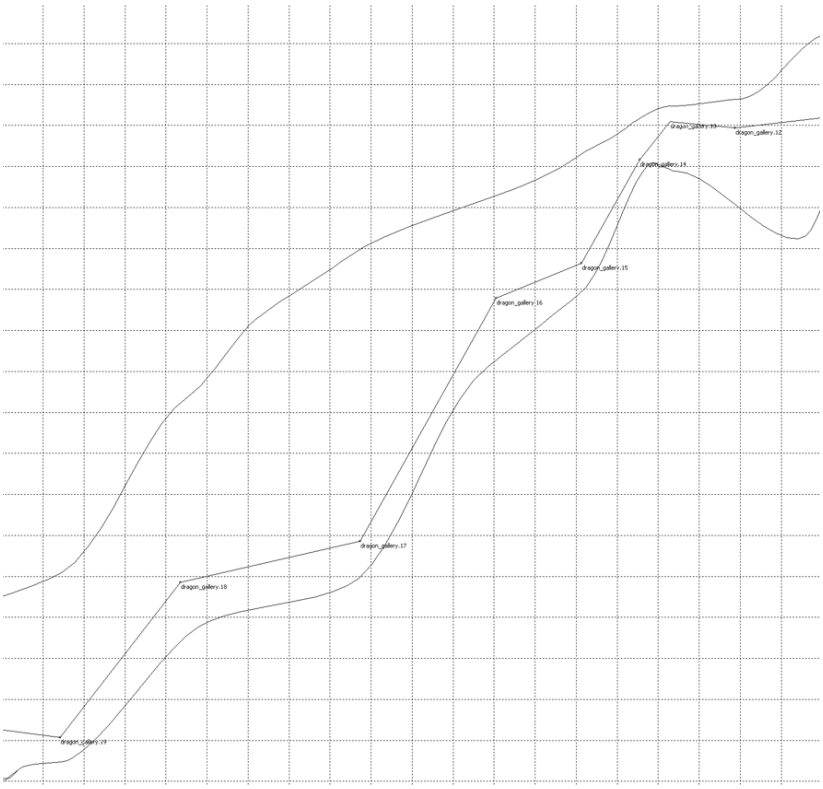


Рис 1.2.3. Вертикальная проекция участка пещеры «Галерея дракона»

Теперь взглянем, как в программе представляются данные о расположении точек.

Рис 1.2.3. Представлении информации о точках пещеры «Снежная»

Первые 2 цифры, показывают какие пикеты соединяются. Если вместо второй цифры стоит прочерк, то задаются координаты отстрела от пикета, обозначенного первой цифрой. Далее идёт информация о азимуте, вертикальном угле и расстоянии до точки.

Глава 2. Решение задачи без отстрелов

Сейчас при спелеосъемке для описания пространства пещеры от каждого пикета производят отстрелы. Но раньше описание происходило гораздо проще и грубее - от каждого пикета рассчитывалось расстояние до каждой из стенок, пола и потолка. Данные метод достаточно условно описывал форму пещеры. Но благодаря его простоте именно с него лучше всего будет начитать поиск характеристик трехмерных моделей пещер, а затем использовать полученные знания для решения более комплексной задачи.

Сначала рассмотрим с какими данными будет вестись работа. В получаемом нами массиве данных содержится: расстояние до следующего пикета, азимут между точками, в которых проводились измерения, вертикальный угол, расстояние от точки измерения до: левой, правой стены, потолка и пола.

```
vector <vector<double>> picket = { {0, 0, 0,1, 1,0 ,0 },
    {5.33,     140, -3.5,     6.5,    2,      3, 0 },
    {11.22,    146, -12.1,    6,      4.3,    3.6, 0 },
    {5.7,      151.5, 12.4,   4.5,    4,      1.5, 0 },
    {12.36,    181,    18.8,  2,      4,      6.5, 0 },
    {12.36,    238,    18.8,  2,      4,      6.5, 0},
    {9.53,     174,    5.9    ,5.5,   2,      0,      2 },
    {6.28,     135, -22.3,    5,      3,      2 ,0 },
    {7.95,     120.5, -5.3,   5.7,    4,      3.5, 0 },
    {6.88,     96.5, -14.1,   4,      3,      11, 0 },
    {5.4,      71, -8,1,      0.5,    1.5, 0 },
    {3.47,     115, -60.2,    1,      1,      1.5, 0 },
    {4.83,     60.5, -8.8,    3,      0,      4, 0 },
    {9.10,     78, -23.8,     1.5,    1.5,    1, 0 },
    {7.84,     191, -12.3,    2,      3,      1, 0 },
    {6.32,     182,    7.8,   2,      2,      2, 0 },
    {8.23,     94,     0,     1.5,    4,      1.5, 0 },
    };
```

Листинг 2.1. Поступающие исходные данные

Координаты первого пикета берём (0,0,0), остальные будем высчитывать.

2.1 Вычисление площади

Сначала рассмотрим функцию **Cave_S**. Она вычисляет площадь пещеры. Получив начальный массив, мы должны найти координаты точек у стен

пещеры. Для этого, сначала вычислим координаты пикетов в декартовой системе координат.

```
dot[0] += picket[1][0] * cos(picket[1][1] * in_gr) * cos(picket[1][2] *
in_gr);
dot[1] += picket[1][0] * sin(picket[1][1] * in_gr) * cos(picket[1][2] *
in_gr);
dot[2] += picket[1][0] * sin((picket[1][2] * in_gr));
Dot.push_back({ dot[0] ,dot[1] ,dot[2] });
```

Листинг 2.1.1. Нахождение координат точки, в которой проводится измерение

Стоит сказать, что данные о азимуте и вертикальном угле, мы получаем в градусах, а функции cos и sin принимают радианы. Поэтому, наши значения мы умножаем на переменную **in_gr**, которая равна $\frac{\pi}{180}$.
Для получения координат точек у правой стены мы возьмем координаты пикета и прибавим к координате «y» расстояние до правой стены. Для другой точки – вычтем от координаты x расстояние до левой стены. И у обоих точек вычтем от координаты «z» расстояние до пола.

```
        dot[0] += picket[i][0] * cos(picket[i][1] * in_gr) *
cos(picket[i][2] * in_gr);
        dot[1] += picket[i][0] * sin(picket[i][1] * in_gr) *
cos(picket[i][2] * in_gr);
        dot[2] += picket[i][0] * sin((picket[i][2] * in_gr));
        Dot.push_back({ dot[0] ,dot[1] ,dot[2] });

        dot1[0] += dot[0] + picket[i][4] * cos(picket[i][1] * in_gr) *
cos(picket[i][2] * in_gr);
        dot2[0] += dot[0] - picket[i][3] * cos(picket[i][1] * in_gr) *
cos(picket[i][2] * in_gr);
        dot1[1] += dot[1] - picket[1][0] * sin(picket[1][1] * in_gr) *
cos(picket[1][2] * in_gr);
        dot2[1] += dot[1] - picket[1][0] * sin(picket[1][1] * in_gr) *
cos(picket[1][2] * in_gr);
        dot1[2] += dot[2] - picket[i][5] * sin((picket[i][2] * in_gr));
        dot2[2] += dot1[2];
        dots.push_back({ dot1 , dot2 });
        dot1[1] += picket[1][0] * sin(picket[1][1] * in_gr) *
cos(picket[1][2] * in_gr);
        dot2[1] += picket[1][0] * sin(picket[1][1] * in_gr) *
cos(picket[1][2] * in_gr);
        dots.push_back({ dot1 , dot2 });
```

Листинг 2.1.2. Нахождение координат точек у стен

Затем, начнём вычислять площадь пещеры. Для этого найдём площадь между каждым пикетом и следующим за ним и сложим.

Площадь между двумя точками измерения представим в виде четырёх угольника. Его мы разделим на 2 треугольника и найдём площадь каждого из них с помощью формулы Герона. Имея координаты 4 точек сначала, найдем вектора, соответствующие сторонам, затем их длины.

```
a = sqrt((dots[i + 1].first[0] - dots[i].second[0])
* (dots[i + 1].first[0] - dots[i].second[0]) +
 (dots[i + 1].first[1] - dots[i].second[1])
 * (dots[i + 1].first[1] - dots[i].second[1]) +
 (dots[i + 1].first[2] - dots[i].second[2])
 * (dots[i + 1].first[2] - dots[i].second[2]));

   b = sqrt((dots[i].first[0] - dots[i].second[0]) *
(dots[i].first[0] - dots[i].second[0]) +
            (dots[i].first[1] - dots[i].second[1])
 * (dots[i].first[1] - dots[i].second[1]) +
            (dots[i].first[2] - dots[i].second[2])
 * (dots[i].first[2] - dots[i].second[2]));

        c = sqrt((dots[i + 1].first[0] - dots[i].first[0])
 * (dots[i + 1].first[0] - dots[i].first[0]) +
            (dots[i + 1].first[1] - dots[i].first[1])
 * (dots[i + 1].first[1] - dots[i].first[1]) +
            (dots[i + 1].first[2] - dots[i].first[2])
 * (dots[i + 1].first[2] - dots[i].first[2]));
```

Листинг 2.1.3. Вычисление сторон первого треугольника

С их помощью мы находим полупериметр и площадь. Поскольку пещеры чаще всего не прямоугольные, чтобы учесть их округлость умножим площадь на коэффициент «К» равный 0.78.

```
p = (a + b + c) * 0.5;
S += sqrt(p * (p - a) * (p - b) * (p - c))*K;
```

Листинг 2.1.4. Нахождение площади первого треугольника

Площадь второго треугольника находится, аналогично. Нам нужно лишь пересчитать переменные «b» и «c». Переменная «a» отвечает за диагональ четырёх угольника, она остаётся прежней.

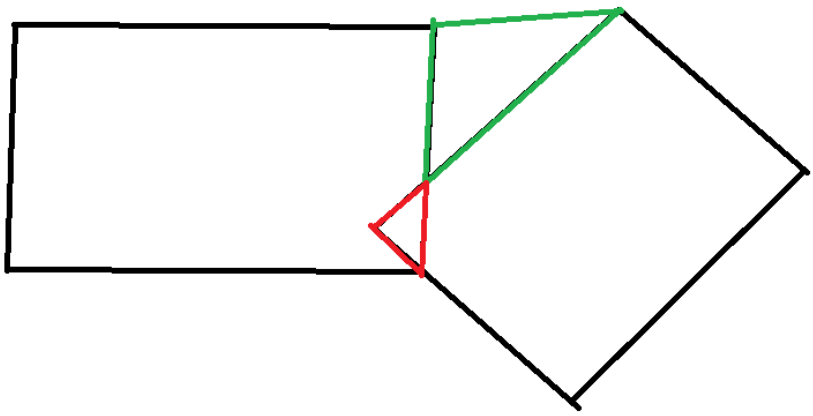
```
b = sqrt((dots[i + 1].first[0] - dots[i + 1].second[0]) * (dots[i +
1].first[0] - dots[i + 1].second[0]) +
            (dots[i + 1].first[1] - dots[i + 1].second[1]) * (dots[i +
1].first[1] - dots[i + 1].second[1]) +
            (dots[i + 1].first[2] - dots[i + 1].second[2]) * (dots[i +
1].first[2] - dots[i + 1].second[2]));

c = sqrt((dots[i + 1].second[0] - dots[i].second[0]) * (dots[i +
1].second[0] - dots[i].second[0]) +
            (dots[i + 1].second[1] - dots[i].second[1]) * (dots[i +
1].second[1] - dots[i].second[1]) +
            (dots[i + 1].second[2] - dots[i].second[2]) * (dots[i +
1].second[2] - dots[i].second[2]));

p = (a + b + c) * 0.5;
S += sqrt(p * (p - a) * (p - b) * (p - c))*K;
```

Листинг 2.1.5. Нахождение площади второго треугольника

Теперь рассмотрим саму схему расчёта площадей. В случае, если следующий пикет находится сильно левее или правее предыдущего у нас возникают ситуации наложения, когда появляются площади, которые мы считаем дважды или наоборот, не считаем во все.



Изображение 2.1.1. Пример наложения площадей

В данном случае площадь красного треугольника мы считаем дважды. А площадь зеленого не считаем совсем.

Всего данных особых случаев 4. Они зависят от того, в какую сторону отдаляется следующий пикет и от разницы в ширине участков между пикетами.

Следующий код показывает пример необходимых вычисления для того, чтобы учесть вышеописанные особые случаи.

```
a1 = sqrt((dots[i].first[0] - dotlast1[0]) * (dots[i].first[0] -
dotlast1[0]) +
                        (dots[i].first[1] - dotlast1[1]) *
(dots[i].first[1] - dotlast1[1]) +
                        (dots[i].first[2] - dotlast1[2]) *
(dots[i].first[2] - dotlast1[2]));

                    b1 = sqrt((dots[i].first[0] - Dot[i/2][0]) *
(dots[i].first[0] - Dot[i / 2][0]) +
                        (dots[i].first[1] - Dot[i / 2][1]) *
(dots[i].first[1] - Dot[i / 2][1]) +
                        (dots[i].first[2] - Dot[i / 2][2]) *
(dots[i].first[2] - Dot[i / 2][2]));

                    c1 = sqrt((Dot[i / 2][0] - dotlast1[0]) * (Dot[i /
2][0] - dotlast1[0]) +
                        (Dot[i / 2][1] - dotlast1[1]) * (Dot[i / 2][1] -
dotlast1[1]) +
                        (Dot[i / 2][2] - dotlast1[2]) * (Dot[i / 2][2] -
dotlast1[2]));

                    p = (a1 + b1 + c1) * 0.5;
                    S -= sqrt(p * (p - a1) * (p - b1) * (p - c1)) * K;

                    a1 = sqrt((dots[i].second[0] - dotlast2[0]) *
(dots[i].second[0] - dotlast2[0]) +
                        (dots[i].second[1] - dotlast2[1]) *
(dots[i].second[1] - dotlast2[1]) +
                        (dots[i].second[2] - dotlast2[2]) *
(dots[i].second[2] - dotlast2[2]));

                    b1 = sqrt((dots[i].second[0] - Dot[i / 2][0]) *
(dots[i].second[0] - Dot[i / 2][0]) +
                        (dots[i].second[1] - Dot[i / 2][1]) *
(dots[i].second[1] - Dot[i / 2][1]) +
                        (dots[i].second[2] - Dot[i / 2][2]) *
(dots[i].second[2] - Dot[i / 2][2]));

                    c1 = sqrt((Dot[i / 2][0] - dotlast2[0]) * (Dot[i /
2][0] - dotlast2[0]) +
                        (Dot[i / 2][1] - dotlast2[1]) * (Dot[i / 2][1] -
dotlast2[1]) +
                        (Dot[i / 2][2] - dotlast2[2]) * (Dot[i / 2][2] -
dotlast2[2]));

                    p = (a1 + b1 + c1) * 0.5;
                    S += sqrt(p * (p - a1) * (p - b1) * (p - c1)) * K;
```

Листинг 2.1.6. Учёт особых случае

Другие особые случаи, вычисляются аналогично. Разница лишь в точках, с которыми производим вычисления.

Затем мы суммируем площади всех треугольников, и получаем площадь всей пещеры.

2.2 Поиск объема пещеры

Объём пещеры будет найден с помощью функции **Cave_V**. Она во многом повторят функции **Cave_S**. Мы всё также обрабатываем исходные данные. Площадь ищем тем же способом.

После нахождения площади первого треугольника, мы ищем среднюю высоту пещеры на этом участке. Затем находим объем первой треугольной призмы. Аналогично находим объём второй.

```
S = sqrt(p * (p - a) * (p - b) * (p - c))*K;
MH = (picket[i][6] + picket[i][5] + picket[i + 1][6] + picket[i + 1][5]) *
0.5;
V += S * MH*0.5;
```

Поскольку пещера не является 4 угольной призмой, учтём это умножив весь объём на коэффициент K.

Листинг 2.2.1. Вычисление объёма треугольной призмы

После суммирования всех призм мы найдем объём пещеры.

Глава 3. Решение Задачи

Решив более простую задачу перейдём к случаю с отстрелами. Как и в более простом случае, мы будем использовать тот же алгоритм получения координат пикетов и поиска объёма пещеры. Ещё можно использовать функции получения координат и поиска площади (но их будет необходимо немного изменить).

3.1 Теоретические сведенья

Определение координат пикетов

«Идея очень простая. У нас есть две точки (мы их далее будем называть пикетами). С помощью рулетки измеряем расстояние между ними. Прикладывая компас к натянутой рулетке, замеряем азимут с одной точки на другую. Прикладывая к рулетке же эклиметр (прибор для измерения вертикальных углов - отвес и шкала в градусах), получаем вертикальный угол. Для горизонтальных каменоломен достаточно расстояния и азимута. Итак, мы получили расстояние между точками L, горизонтальный (a) и вертикальный (b) углы. Если мы примем первую точку за нулевую (x=0, y=0, z=0), а ось "Y" направим на магнитный север, то мы легко получим координаты второй точки. Для двумерного случая каменоломен:

X=L*cos(a)

Y=L*sin(a)

Далее измеряем те же величины между второй и третьей точками, а координаты третьей точки получаем прибавлением к координатам второй точки вновь вычисленных X и Y. Таким образом мы получаем цепочку точек с известными координатами. Отстроив их на миллиметровке, мы получим "ход", который описывает нашу пещеру. А если мы в каждой точке измеряли расстояние от пикета до правой и левой стенок, то мы легко обрисуем наш ход и получим изображение еще и ширины штреков.

Для трехмерного случая природных пещер все несколько иначе. Казалось бы, мы должны получить трехмерные координаты точек:

X=L*cos(a)*cos(b)

Y=L*sin(a)*cos(b)

Z=L*sin(b)»

Определение объёма пещеры

«Объем пещеры (V, куб. м) определяется по следующей формуле: V=K•S•H, где К - коэффициент, учитывающий характер поперечного сечения полости (для прямоугольного 1,0; для овального и круглого 0,78; для треугольного сечения - 0,5); 5 - площадь пола, кв. м; Н - средняя высота по оси хода, м.»

Ориентированная площадь треугольника

Для вычисления площади пола и при поиске объёма будет использована ориентированная площадь треугольника.

«Ориентированной площадью ориентированного треугольника ABC мы будем называть число, абсолютная величина которого равна площади треугольника (неориентированного) с вершинами A, B, C; ориентированную площадь мы будем считать положительным числом, если треугольник ориентирован против часовой стрелки, и отрицательным числом, если треугольник ориентирован по часовой стрелке.

Ориентированная площадь треугольника, образованного векторами $\underset{a}{\rightarrow}$ и $\underset{b}{\rightarrow}$ - $S_{\Delta} = \frac{1}{2} * \left[\underset{a}{\rightarrow} * \underset{b}{\rightarrow}\right]$»

Для нахождение объёма пещеры мы будем представлять некоторые её части в виде усечённых пирамид или эллипсоидов. Формулы для вычисления объёма этих фигур будут приведены далее.

Объём усеченной пирамиды

$V = \frac{1}{3} * H * (S_1 + \sqrt{S_1 * S_2} + S_2)$;

H – высота пирамиды;

S_1,S_2 – площади оснований.

Объём эллипсоида

$V = \frac{4}{3} * \pi * a*b*c;$

Площадь эллипса - $\pi * a*b;$

3.2 Обработка пикетов

Перед тем, как искать площадь и объём необходимо обработать пикеты, получив их координаты.

С этой целью была написана функция ***Dots_interpenter***. На вход она получает кубический вектор с данными о пикетах, а на выходе получаем другой кубический вектор.

Для получения координат пикетов и отстрелов, будут использованы формулы, упомянутые в первом разделе. При нахождении координат следующего пикета мы используем эти формулы и прибавляем результат к координатам прядущего. Таким же образом мы определяем координаты отстрелов для каждого пикета.

Стоит сказать, что нам нужны координаты только отстрелов, и данные о пикетах сохраняться не будут.

```
for (int i = 0; i < size(picket); i++)
    {
        dot2[0] +=  picket[i][0][0] * cos(picket[i][0][1] * in_gr) *
cos(picket[i][0][2] * in_gr);
        dot2[1] += picket[i][0][0] * sin(picket[i][0][1] * in_gr) *
cos(picket[i][0][2] * in_gr);
        dot2[2] += picket[i][0][0] * sin((picket[i][0][2] * in_gr));


        for (int j = 1; j < size(picket[i]); j++)
        {
            dot[0] = dot2[0] + picket[i][j][0] * cos(picket[i][j][1] *
in_gr) * cos(picket[i][j][2] * in_gr);
            dot[1] = dot2[1] + picket[i][j][0] * sin(picket[i][j][1] *
in_gr) * cos(picket[i][j][2] * in_gr);
            dot[2] = dot2[2] + picket[i][j][0] * sin((picket[i][j][2] *
in_gr));
            Tdot.push_back(dot);
        }
        Dot.push_back(Tdot);
        Tdot.clear();
    }
```

Листинг 3.2.1. Нахождение координат пикетов и отстрелов

3.3. Нахождение площади

Теперь, перейдем к нахождению площади. Для этого используется несколько методов. Подробнее рассмотрим каждый их них.

Метод 1.

Данный метод является наиболее простым в реализации, также он является самым неточным. Первый метод был реализован в функции ***Cave_S***. На вход она принимает кубический вектор и возвращает число.

Принцип работы функции заключается в:

1) Нахождении точек, которые максимально отдалены друг от друга по координате «y», у каждого пикета
2) Изменение значение координаты «x» у каждой точки на минимальное значение для каждого пикета
3) Вычисление площади получившегося четырёхугольника с помощью метода Герона

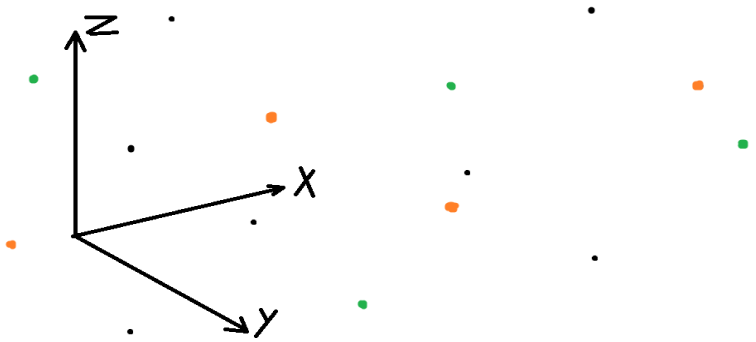


Рис 3.3.1. Пример расположения точек. Зеленым обозначены «экстремумы» по координате «y». Оранжевым – по «x»

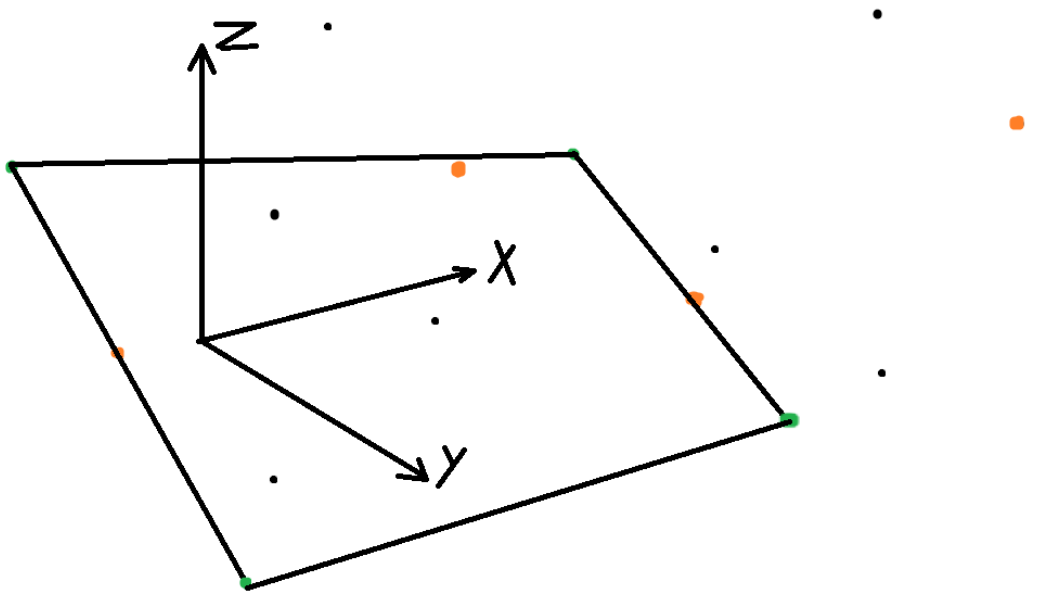


Рис 3.3.2. Перемещение точек и выделение четырёхугольника, площадь которого будет вычислена

Это верно для всех пикетов, кроме 2 последних. В их случае 2 пункт будет изменен, и для последнего пикета мы будем искать точку максимальную по «x». Это было сделано для того, чтобы избежать наложений площадей.

Также необходимо уточнить, что при нахождении площади пола разница в высоте между точками игнорируется.

Подробнее рассмотрим каждый из шагов. Для нахождения точек, максимально удалённых по координате «y», мы находим пары точек с максимальным и минимальным значением по данной координате. Сейчас и в следующих случаях для решения подобных задач будет применятся функция ***min_elm*** и ***max_elm***. Данные функции принимают вектор точек и число, обозначающее, по какой координате будет вестись поиск. (0 – по «x», 1 – по «y» и т.д.)

Для изменения значений, мы с помощью функции ***min_elm*** найдем точку с самым маленьким значением координаты «x» в пикете и изменим значение координат ранее найденных нами точек на данное.

При вычислении площади четырёхугольника воспользуемся функцией ***area_of_triangle_Ger***. Чтобы с её помощью вычислить площадь нужно разделить нашу фигуру на 2 треугольника. В первый входят следующие точки: максимальная и минимальная из пары текущих отстрелов, минимальная из

следующих. Во второй: максимальная и минимальная из пары следующего отстрела, максимальная из текущего.

Таким образом площадью пещеры будет сумма площадей всех этих треугольников.

```
T S = 0;
    vector<T> a, b,c,d,e,f;

    for (int i = 0; i < Dot.size() - 2; i++)
    {
        a = max_elm(Dot[i], 1);
        b = min_elm(Dot[i], 1);
        c = max_elm(Dot[i+1], 1);
        d = min_elm(Dot[i+1], 1);
        e = min_elm(Dot[i], 0);
        f = min_elm(Dot[i + 1], 0);
        a[0] = b[0] = e[0];
        c[0] = d[0] = f[0];
        S += area_of_triangle_Ger(b, a, d);
        S += area_of_triangle_Ger(d,c, a);

    }
    a = max_elm(Dot[Dot.size() - 2], 1);
    b = min_elm(Dot[Dot.size() - 2], 1);
    c = max_elm(Dot[Dot.size() - 1], 1);
    d = min_elm(Dot[Dot.size() - 1], 1);
    e = min_elm(Dot[Dot.size() - 2], 0);
    f = max_elm(Dot[Dot.size() - 1], 0);
    a[0] = b[0] = e[0];
    c[0] = d[0] = f[0];
    S += area_of_triangle_Ger(b, a, d);
    S += area_of_triangle_Ger(d, c, a);

    return S;
```

Листинг 3.3.1. Нахождение площади пещеры 1 методом

Метод 2. Построение предположительного контура пола и поиск его площади с помощью метода Герона

Суть метода заключается в попытке построить контур пола. Для этого мы сортируем отстрелы вокруг каждого пикета следующим образом:

A. Находим 4 точки «экстремума» (точки у которых координат «x» или «y» максимальна или минимальна) и нумеруем их (максимальная по «x» - 0, максимальная по «y» - 1, минимальная по «x» - 2, минимальная по «y» - 3). Данные точки будут первыми у соответствующих векторов.
B. Далее мы причисляем оставшиеся точки к этим векторам по следующему принципу:
    a) Сначала происходит первичная сортировка. Мы смотрим к какой из точек «экстремума» наша точка ближе и отправляем в соответствующий вспомогательный вектор. Для этого используется функция ***Sort_one***.
    b) Во время вторичный сортировки мы смотрим на расстояние до соседних точек «экстремума» (если рассматриваем точку из 0 вектора, то смотрим на расстояние до 3 и 1, если из 1 – то до 0 и 2, и т.д.). Если точка находится ближе к предыдущему «экстремуму», то она окончательно переносится в вектор предыдущего, если она ближе к следующему – то перемещается в вектор текущего «экстремума» (например, точка из 0 вспомогательного вектора находится ближе к 3 точке «экстремума», чем к 1, то она оказывается в 3 векторе. Если бы она была ближе к 1, то была бы прикреплена к 0 вектору)
C. Затем мы сортируем точки внутри каждого вектора так, чтобы сначала шёл «экстремум», а затем остальные точки по мере их удаления от «экстремума». Это достигается применением функции ***Sort_lenghts***.

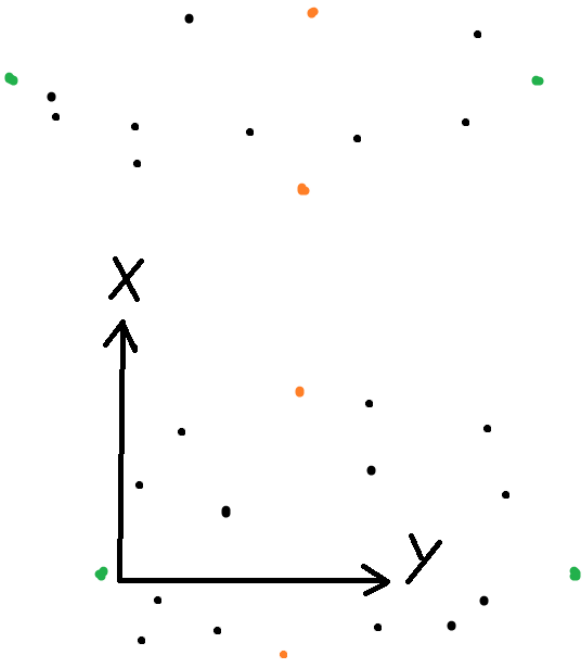


Рис 3.3.4. Пример расположения точек

Затем, мы берем вектора 1 и 2 (они описывают нижнюю половину нашего многоугольника) и вычисляем площадь получившейся фигуры. Делаем мы это с помощью функции ***area_of_triangle_Ger***. Для этого берем точки в следующем порядке: первая точка из вектора, следующая точка из того же вектора и «экстремум» следующего вектора. И последний шаг прибавить площадь треугольника, построенного на 3 «экстремумах» (мы берем максимальную и минимальную точку по «y» и минимальную точку по «x»).

Также мы находим площадь четырёхугольника, построенного на «экстремумах» по «y» для обоих пикетов.

К получившейся площади четырёхугольника мы прибавляем площадь нижней половины многоугольника текущего пикета, и отнимаем площадь следующего (так как она уже была почитана при вычислении площади четырёхугольника).

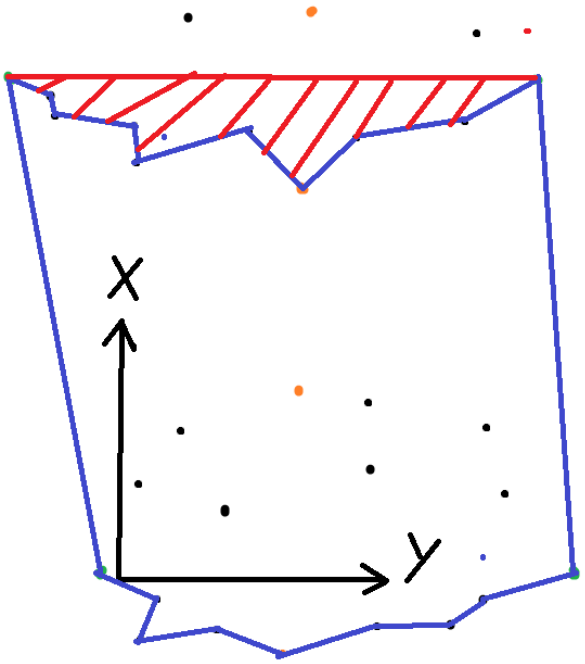


Рис 3.3.5. Вычисление площади пола вторым методом. Красным обозначена область, которую мы будем вычитать

Аналогично с первым методом, этот применимо для всех пикетов, кроме двух последних. Разница в том, что мы не отнимаем площадь нижнего многоугольника, а прибавляем площадь верхнего.

```
            a = max_elm(Dot[i], 0);
            b = min_elm(Dot[i], 0);
            c = max_elm(Dot[i], 1);
            d = min_elm(Dot[i], 1);
             dots1 = { {a},{c}, {b}, {d} };
```

Листинг 3.4.2. Нахождение точек экстремума

```
Numbers = { {},{},{},{} };
            dots1 = { {a}, {c}, {b}, {d} };
            for (int j = 0; j < Dot[i].size(); j++)
            {
                if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c &&
Dot[i][j] != d)
                    Numbers[Sort_one(a, b, c, d, Dot[i][j],
lenghtS)].push_back(Dot[i][j]);

            }
            for (int j = 0; j < Numbers[0].size(); j++)
            {
                if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
                {
                    dots1[3].push_back(Numbers[0][j]);
                }
            }
            for (int j = 0; j < Numbers[1].size(); j++)
            {
                if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
                {
                    dots1[1].push_back(Numbers[1][j]);
                }
            }
            for (int j = 0; j < Numbers[2].size(); j++)
            {
                if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
                {
                    dots1[2].push_back(Numbers[2][j]);
                }
                else
                {
                    dots1[1].push_back(Numbers[2][j]);
                }
            }
            for (int j = 0; j < Numbers[3].size(); j++)
            {
                if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
                {
                    dots1[3].push_back(Numbers[3][j]);
                }
                else
                {
                    dots1[2].push_back(Numbers[3][j]);
                }
            }
            Numbers.clear();
```

Листинг 3.3.3. Первичная и вторичная сортировка всех точек

```
      dots1 = Sort_lenghts(dots1, lenghtS);
            for (int i = 0; i < dots1[1].size()-1; i++)
            {
                S += area_of_triangle_Ger(c,
dots1[1][i+1], dots1[1][i]);
            }
            S += area_of_triangle_Ger(c, b, d);
            for (int i = 0; i < dots1[2].size()-1; i++)
            {
                S += area_of_triangle_Ger(d, dots1[2][i
+ 1], dots1[2][i]);
            }

            dots2 = Sort_lenghts(dots2, lenghtS);
            for (int i = 0; i < dots2[1].size() - 1;
i++)
            {
                S -= area_of_triangle_Ger(c, dots2[1][i
+ 1], dots2[1][i]);
            }
            S -= area_of_triangle_Ger(c, b, d);
            for (int i = 0; i < dots2[2].size() - 1;
i++)
            {
                S -= area_of_triangle_Ger(d, dots2[2][i
+ 1], dots2[2][i]);
            }
```

Листинг 3.3.4. Сортировка внутри каждого вектора и вычисление площади для нижних многоугольников

Метод 3. Построение контура и использование метода «Ориентированных площадей»

Этот метод сильно схож со вторым методом. Отличие заключается в том, что для поиска площадей нижних половин многоугольников мы будем использовать не метод Герона, а ориентированные площади.

Для этого мы точно также сортируем все точки. Но нулевой вектор будет удалён полностью, а 3 вектор состоять только из «экстремума». Вычисление площади происходит с помощью функции ***S_nap***.

В этой функции вычисление площади каждого треугольника происходит с помощью функции ***Vector_P_Nap***.

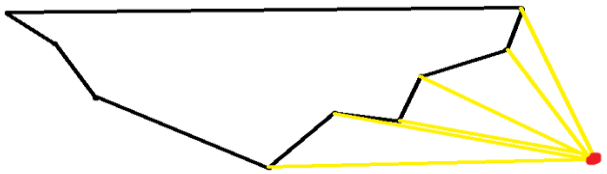


Рис 3.3.6. Пример вычисления площади 3 методом. Красным обозначена точка, образующая все треугольники (метод не зависит от координат этой точки). Желтым обозначены треугольники с отрицательной площадью.

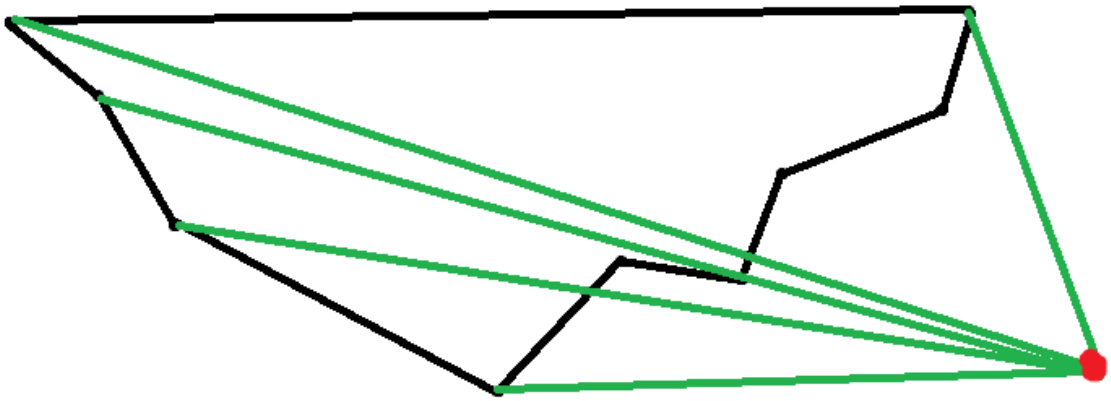


Рис 3.3.7. Пример вычисления площади 3 методом. Зелёным обозначены треугольники с положительной площадью.

В результате, сумма всех треугольников будет равна площади, искомой нам фигуры.

```
 T S = 0;


    for (int i = 0; i < dots.size() - 1; i++)
    {
        for (int j = 0; j < dots[i].size() - 1; j++)
        {
            S += Vector_P_Nap(dots[i][j], dots[i][j +
1]);
        }
        S += Vector_P_Nap(dots[i][dots[i].size() - 1],
dots[i +1 ][0]);
    }

    for (int j = 0; j < dots[dots.size() - 1].size() -
1; j++)
    {
        S += Vector_P_Nap(dots[dots.size() - 1][j],
dots[dots.size() - 1][j + 1]);
    }
    S += Vector_P_Nap(dots[dots.size() -
1][dots[dots.size() - 1].size() - 1], dots[0][0]);

    return 0.5*S;
```

Листинг 3.3.5. Поиск площади всей фигуры

```
    T Ans = 0;
    Ans += A[0]*B[1] - B[0] * A[1];
    return Ans;
```

Листинг 3.3.6. Поиск площади для треугольников составляющих фигуру

Метод 4. Нахождение площади через векторное произведение

Данный метод практически идентичен 3. Разница заключается в том, что мы будем по-другому вычислять площадь нижних многоугольников. Мы будем делать это через векторное произведение. Вектора будут строится на 3 точках: 2 из них описывают контур, а 3 будет одна и та же для всех точек одного пикета. Она будет находится посередине между экстремумами по «y».

Для поиска площади всей фигуры используется функция ***S_nap2***, а для площади отдельных треугольников - ***Vector_P_D***.

```
T S = 0;

    vector <T> T1{ 0,0 };
    vector <T> T2{ 0,0 };

    for (int i = 0; i < dots.size() - 1; i++)
    {
        for (int j = 0; j < dots[i].size() - 1; j++)
        {
            T1[0] = dots[i][j][0] - V[0]; T1[1] = dots[i][j][1] - V[1];
            T2[0] = dots[i][j + 1][0] - V[0]; T2[1] = dots[i][j + 1][1] -
V[1];
            S += Vector_P_D(T1, T2);
        }
        T1[0] = dots[i][dots[i].size() - 1][0] - V[0]; T1[1] =
dots[i][dots[i].size() - 1][1] - V[1];
        T2[0] = dots[i + 1][0][0] - V[0]; T2[1] = dots[i + 1][0][1] -
V[1];
        S += Vector_P_D(T1, T2);
    }

    for (int j = 0; j < dots[dots.size() - 1].size() - 1; j++)
    {
        T1[0] = dots[dots.size() - 1][j][0] - V[0]; T1[1] =
dots[dots.size() - 1][j][1] - V[1];
        T2[0] = dots[dots.size() - 1][j + 1][0] - V[0]; T2[1] =
dots[dots.size() - 1][j + 1][1] - V[1];
        S += Vector_P_D(T1, T2);
    }
    T1[0] = dots[dots.size() - 1][dots[dots.size() - 1].size() - 1][0] -
V[0]; T1[1] = dots[dots.size() - 1][dots[dots.size() - 1].size() - 1][1] -
V[1];
    T2[0] = dots[0][0][0] - V[0]; T2[1] = dots[0][0][1] - V[1];
    S += Vector_P_D(T1, T2);

    return 0.5 * S;
```

Листинг 3.3.7. Поиск площади всей фигуры

```
T Ans = 0;
 Ans += abs(A[0] * B[1] - B[0] * A[1]);
 return Ans;
```

Листинг 3.3.8. Поиск площади для треугольников составляющих фигуру

3.4. Нахождение объёма

Для нахождения объёма пещеры также было написано несколько методов. Перейдём к их подробному рассмотрению.

Метод 1. Представление пространства в виде усечённой пирамиды

Для поиска объёма мы будем представлять пространство между 2 пикетами в виде усеченной пирамиды. В качестве оснований будем брать многоугольники, образуемые стенами, полом и потолком.

Искать их площадь будем с помощью ***S_nap_V***. Представление многоугольников схоже с тем, что было использовано в 3 методе нахождения площади, но на 0 и 2 векторах точки будут выбраны и отсортированы не по координате «x», а по координате «z».

В качестве высоты возьмем расстояние от минимальных точек по координате «x», спроецированных на плоскость XY.

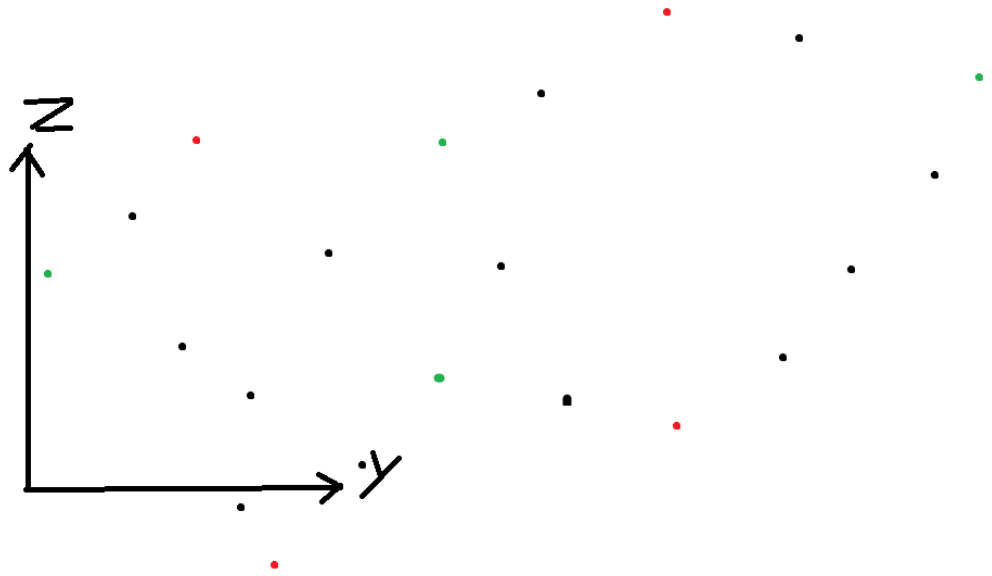


Рис 3.4.1. Пример расположения точек для 1 метода поиска объёма. Красный – «экстремум» по «z». Зеленый – по «y»

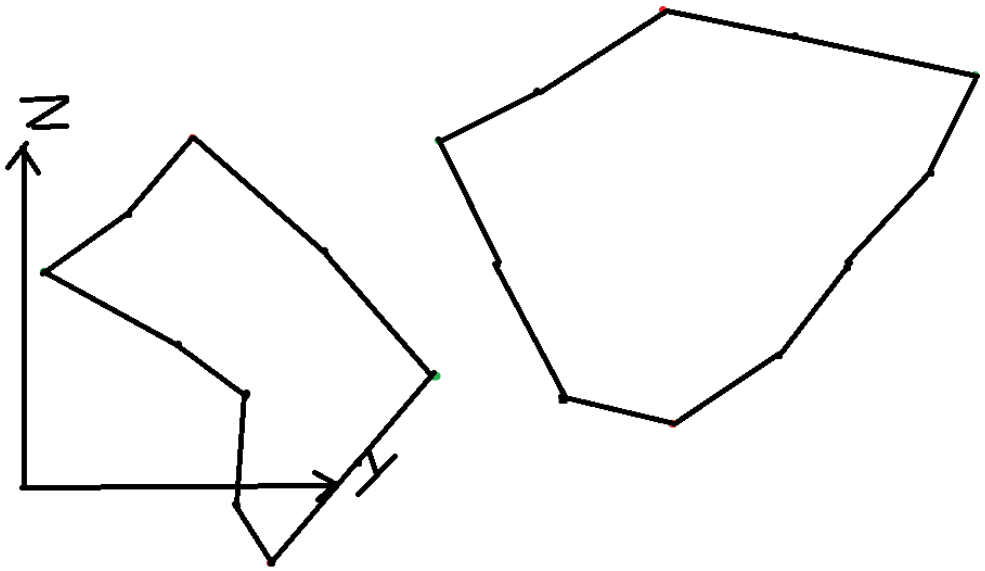


Рис 3.4.2. Построение оснований усечённой пирамиды

Вычисления, производимые с последними двумя пикетами, вновь будут отличаться. Высотой будет считаться расстоянии от минимальной (у предпоследнего пикета) и максимальной (у последнего) точек по координате «x».

```
                    Numbers = { {},{},{},{} };
                    a = max_elm(Dot[i], 2);
                    b = min_elm(Dot[i], 2);
                    c = max_elm(Dot[i], 1);
                    d = min_elm(Dot[i], 1);
                    dots1 = { {a}, {c}, {b}, {d} };
                    dots1 = dots_original_V(dots1, Dot[i]);
                    for (int j = 0; j < Dot[i].size(); j++)
                    {
                        if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c &&
Dot[i][j] != d)
                            Numbers[Sort_one(a, b, c, d, Dot[i][j],
lenghtV)].push_back(Dot[i][j]);

                    }
                    for (int j = 0; j < Numbers[0].size(); j++)
                    {
                        if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
                        {
                            dots1[0].push_back(Numbers[0][j]);
                        }
                        else
                        {
                            dots1[3].push_back(Numbers[0][j]);
                        }
                    }
                    for (int j = 0; j < Numbers[1].size(); j++)
                    {
                        if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
                        {
                            dots1[1].push_back(Numbers[1][j]);
                        }
                        else
                        {
                            dots1[0].push_back(Numbers[1][j]);
                        }
                    }
                    for (int j = 0; j < Numbers[2].size(); j++)
                    {
                        if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
                        {
                            dots1[2].push_back(Numbers[2][j]);
                        }
                        else
                        {
                            dots1[1].push_back(Numbers[2][j]);
                        }
                    }
                    for (int j = 0; j < Numbers[3].size(); j++)
                    {
                        if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
                        {
                            dots1[3].push_back(Numbers[3][j]);
                        }
                        else
                        {
                            dots1[2].push_back(Numbers[3][j]);
                        }
                    }
                }
```

Листинг 3.4.1. Сортировка точек

Метод 2. Представление пространства в виде эллипсоида

В основе данного метода лежит сходство контура пола с эллипсом. Тогда объём пространство между пикетами, описываемый отстрелами можно представить в виде половины эллипсоида.

Площадь основанию мы получаем из результата работы функции ***Cave_S2_2***, которая ищет площадь 3 методом.

Имея площадь эллипса остаются найти значение 3 полуоси. За него примем среднее арифметическое между высотами пикетов.

Также, дабы не возникало наложений, при вычислении объёма между каждой парой пикетов необходимо вычитать объём «полусферы» (основание многоугольник, основанный на обстрелах следующего пикета).

```
            Temp = { Dot[i], Dot[i + 1] };
            S = Cave_S2_2(Temp);
            C = (sqrt((max_elm(Dot[i], 2)[2] -
min_elm(Dot[i], 2)[2]) * (max_elm(Dot[i], 2)[2] -
min_elm(Dot[i], 2)[2])) +
                sqrt((max_elm(Dot[i + 1], 2)[2] -
min_elm(Dot[i + 1], 2)[2]) * (max_elm(Dot[i + 1], 2)[2]
- min_elm(Dot[i + 1], 2)[2]))) *
                0.5;
            V += coef * S * C;
```

Листинг 3.4.2. Вычисление объёма эллипсоида

```
                a = max_elm(Dot[i + 1], 0);
                b = min_elm(Dot[i + 1], 0);
                c = max_elm(Dot[i + 1], 1);
                d = min_elm(Dot[i + 1], 1);
                dots2 = { {a}, {c}, {b}, {d} };

                Numbers = { {},{},{},{} };
                for (int j = 0; j < Dot[i + 1].size(); j++) //Переписать
                {
                    if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i +
1][j] != c && Dot[i + 1][j] != d)
                        Numbers[Sort_one(a,   b,   c,   d,   Dot[i   +   1][j],
lenghtS)].push_back(Dot[i + 1][j]);

                }
                for (int j = 0; j < Numbers[0].size(); j++)
                {
                    if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
                    {
                        dots2[0].push_back(Numbers[0][j]);
                    }
                    else
                    {
                        dots2[3].push_back(Numbers[0][j]);
                    }
                }
                for (int j = 0; j < Numbers[1].size(); j++)
                {
                    if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
                    {
                        dots2[1].push_back(Numbers[1][j]);
                    }
                    else
                    {
                        dots2[0].push_back(Numbers[1][j]);
                    }
                }
                for (int j = 0; j < Numbers[2].size(); j++)
                {
                    if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
                    {
                        dots2[2].push_back(Numbers[2][j]);
                    }
                    else
                        dots2[1].push_back(Numbers[2][j]);
                }
                for (int j = 0; j < Numbers[3].size(); j++)
                {
                    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
                    {
                        dots2[3].push_back(Numbers[3][j]);
                    }
                    else
                        dots2[2].push_back(Numbers[3][j]);
                }
                Numbers.clear();
                dots2 = Sort_lenghts(dots2, lenghtS);
                S = S_nap(dots2);
```

Листинг 3.4.3. Вычисление площади сферы

```
        C = (sqrt((max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2])
* (max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]))) *
            0.5;
        V -= coef * S * C;
```

Листинг 3.4.4. Вычисление объёма сферы

У методов 1 и 2 есть вариации, которые отличаются методом нахождения площади. В них площадь находится через векторное произведение. Совершенное другой подход используется в методе 3 и его варианиях.

Метод 3. Полная триангуляция.

В данном методе на основании всех точек отстрелов мы проводим триангуляцию (разбитие фигуры на треугольники) всего пространства участка пещеры. Эти треугольники будут основаниями пирамид объём, которых мы будем искать. Вершиной пирамид будет точка посередине пространства пещеры. Объём будем искать с помощью смешанного произведения векторов. Это выполняется функциями ***Sorted_dots*** и ***Determinant*** (сам расчет численного значения объёма происходит в функции ***Determinant***, а функция ***Sorted_dots*** преобразует точки, на которых строится пирамида в вектора).

Теперь рассмотрим сам процесс триангуляции. Необходимо рассмотреть 2 случая того, как проходит триангуляция. В 1, тривиальном случае, у каждого пикета 4 отстрела. Тогда каждый отстрел помещается в одно из 4 категорий: набольшее или наименьшее значение координаты «z», набольшее или наименьшее значение координаты «y». Из них составляется массив. В нём они перечисленным в следующем порядке: max z (0), max y(1), min z(2), min y(3).

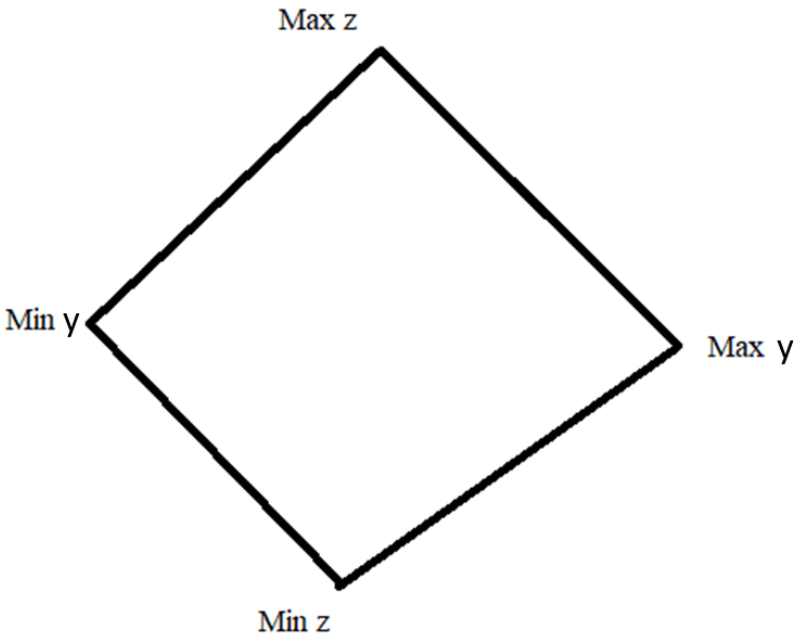


Рисунок 3.4.3. Пример сортировки отстрелов

Затем отстрелы обоих пикетов объединяются в треугольники по следующему принципу: берем i точку из первого массива отстрелов, i точку из второго

массива отстрелов, i+1 точку из первого массива. В таком порядке передаём их в функцию ***Sorted_dots*.**

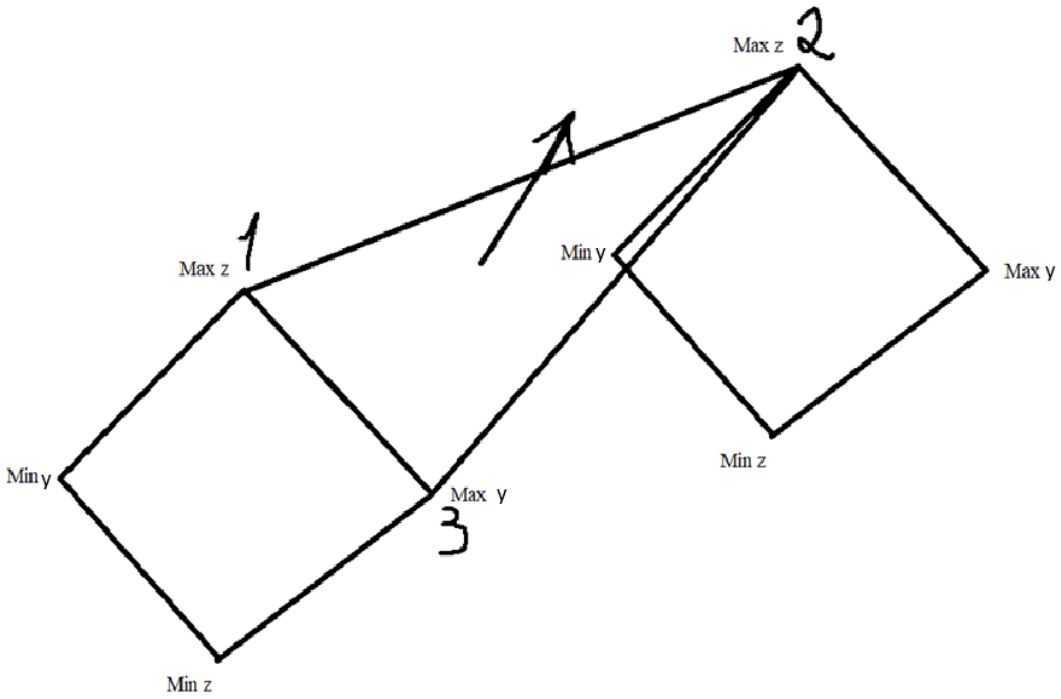


Рисунок 3.4.4. Составление первого треугольника

Далее мы рисуем следующий треугольник для этой стороны фигуры. Мы берем: i точку из второго вектора (назовём её точкой A), i+1 из первого вектора (точка B) и i+1 из второго вектора (точка C).

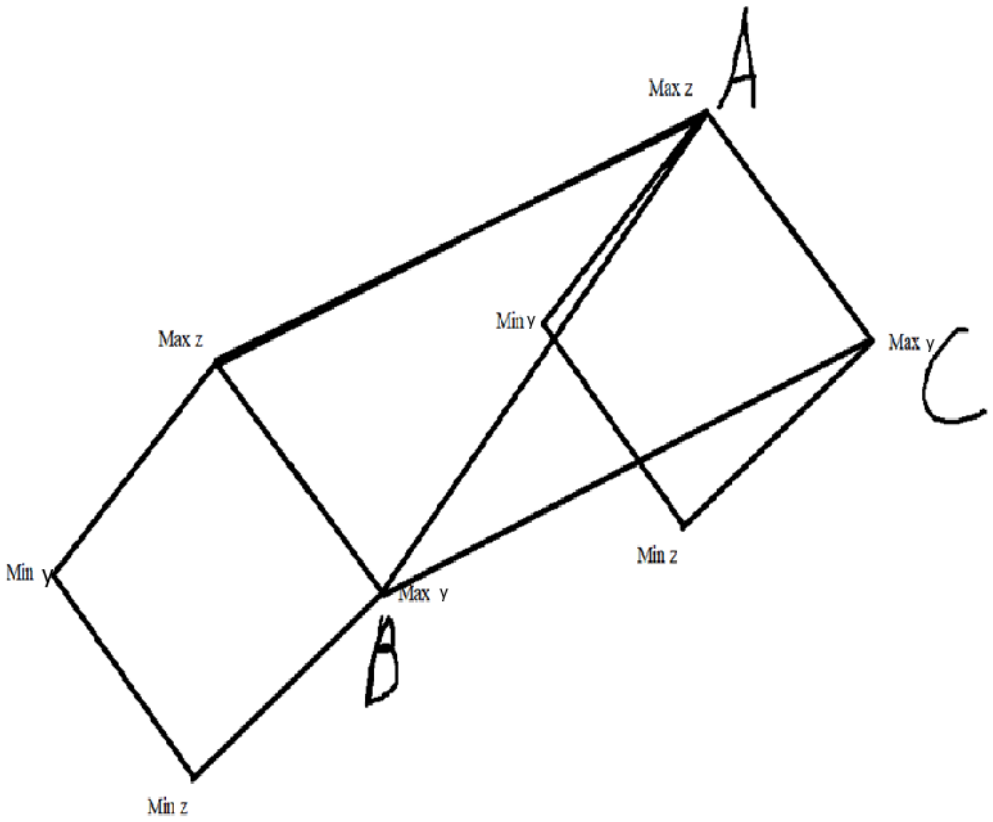


Рисунок 3.4.5. Составление второго треугольника

Таким образом мы разобьём на треугольники каждую из 4 боковых сторон фигуры.

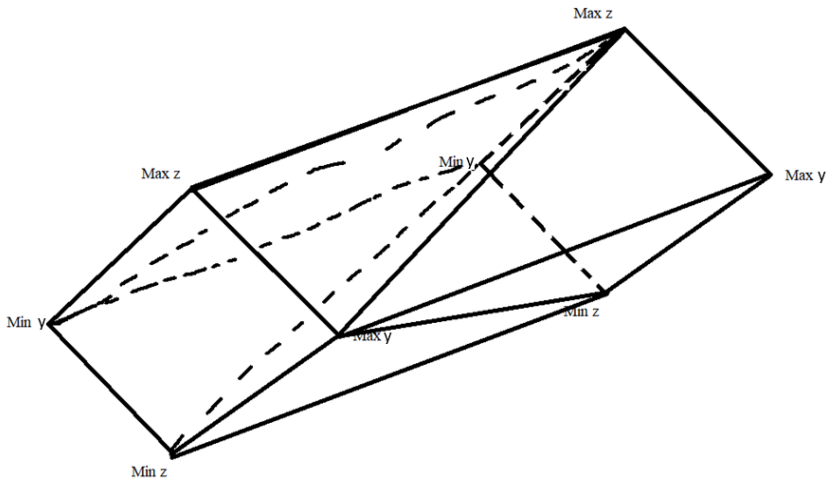


Рисунок 3.4.6. Пример триангуляции боковых сторон

Основания разбиваются на треугольник следующим образом: max «z», min «y», min «z» - для первого треугольника. max «z», min «z» , max «y». Таким образом разбиваются оба основания.

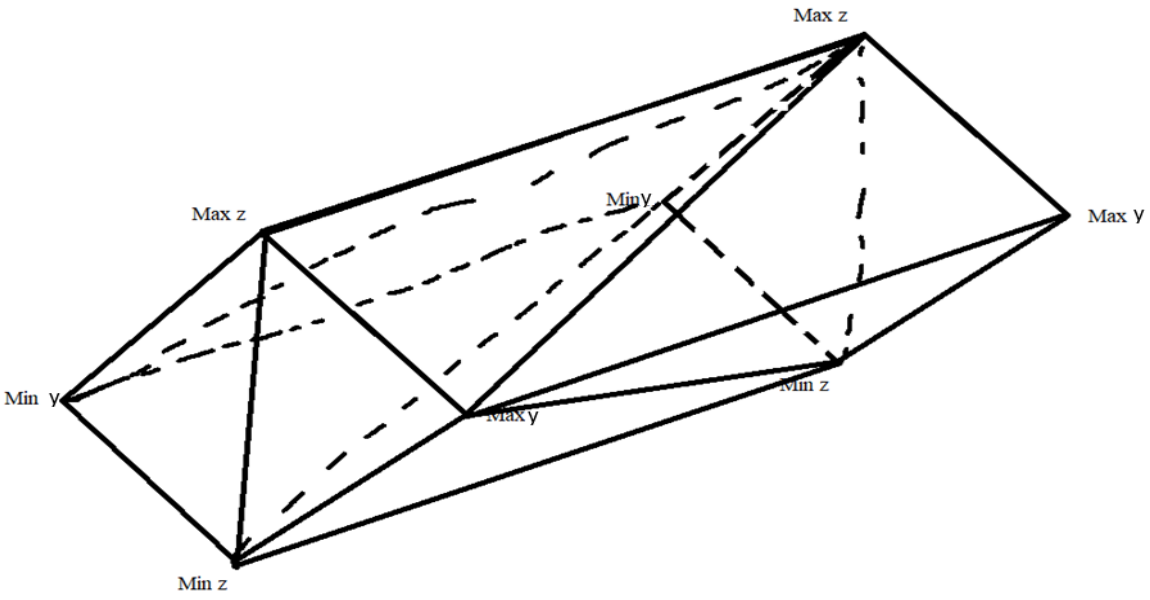


Рисунок 3.4.7. Пример полной триангуляции. Простой случай

Данный вариант триангуляции используется в 2 функция. В первой мы пытаемся построить триангуляцию только по 4 точкам от каждого пикета (ожидается, что полученные значения будут сильно меньше реальных). Во второй мы пытаемся экстраполировать полученную триангуляцию (фактически, мы изменяем значение всех точек по «x» на минимальные для каждого пикета).

Второй случай, когда у одного из пикетов (или у обоих) количество отстрелов больше 4.

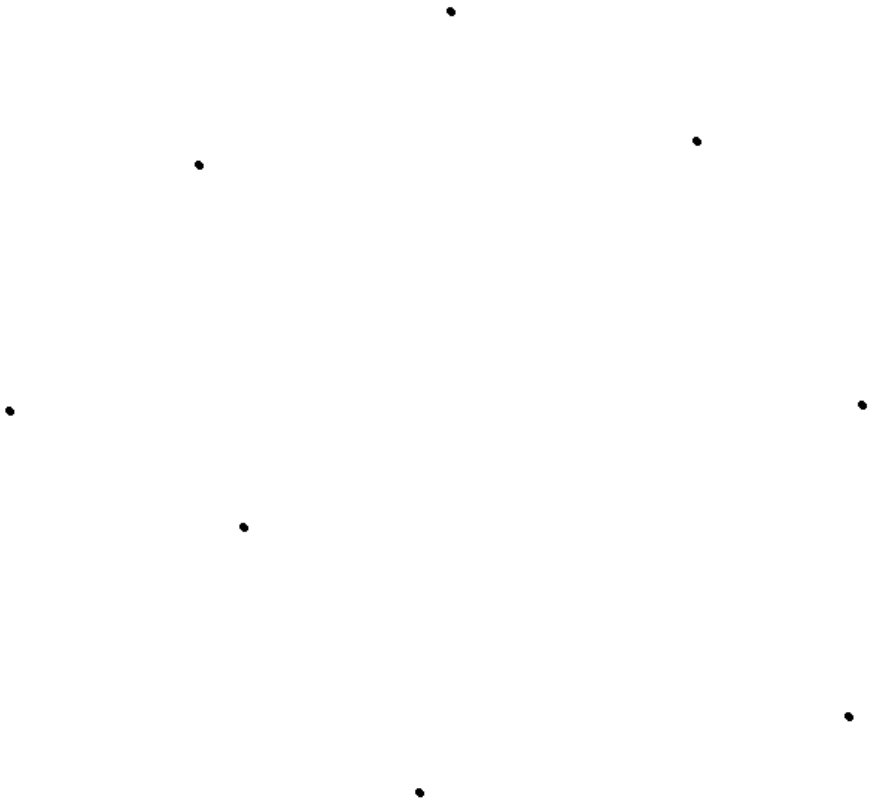


Рисунок 3.4.9. Пример начального расположения точек

Тогда, мы, как и в первом случае, выбираем 4 точки и категорируем их.

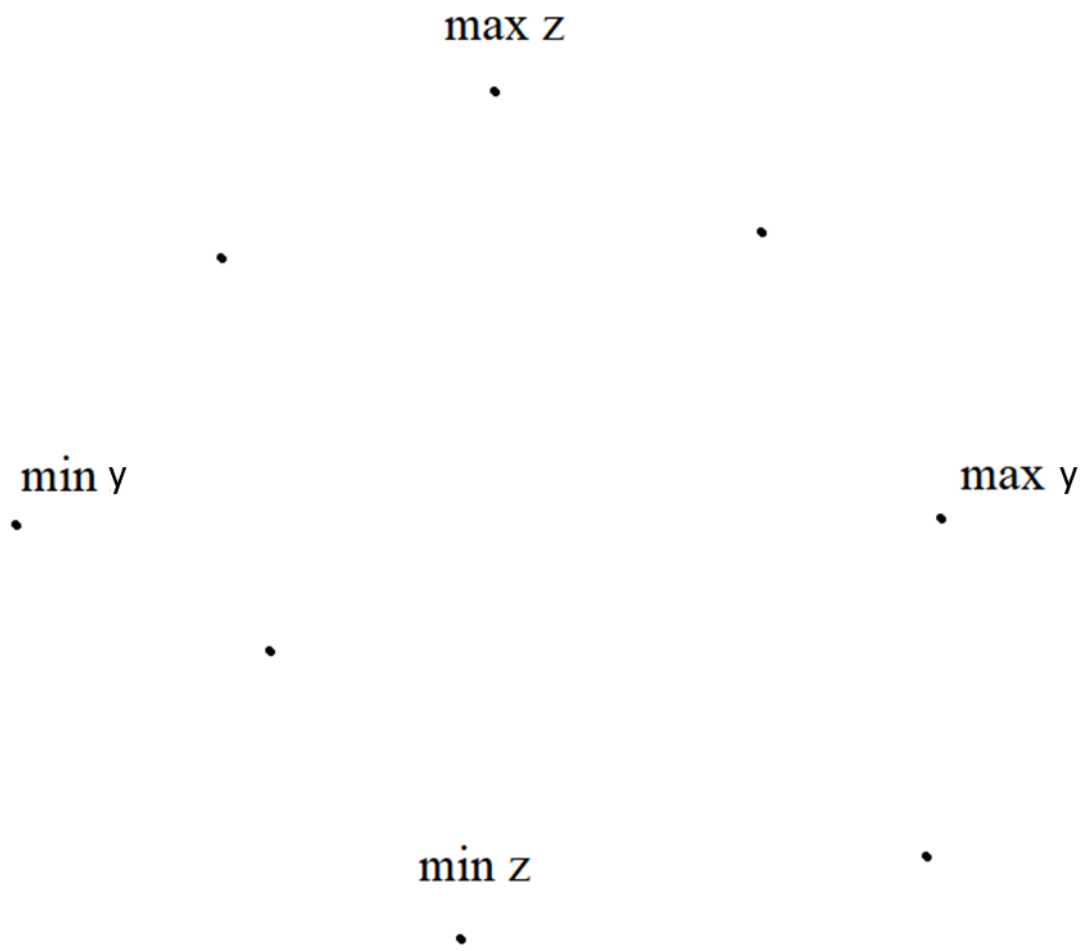


Рисунок 3.4.10. Выделение опорных точек

Остальные точки, мы будем определять в одну из этих групп, аналогично с тем, как мы делали это в методе «Усеченной пирамиды».

В результате, получим категорированные группы для каждого из пикетов.

Рисунок 3.4.11. Точки с группированные для обоих пикетов

Теперь рассмотрим триангуляцию сторон. Если в текущей группу у каждого пикета только один элемент, то она проходит также, как и в тривиальном случае. Если же хоть в одной из групп точек больше одной (для наглядности, взглянем на случай, когда в группе у каждого пикета точек несколько), то мы берем первую точку і группы у первого пикета, первую точку і группы у второго пикета и вторую точку і группы у первого пикета и передаём в функцию**.**

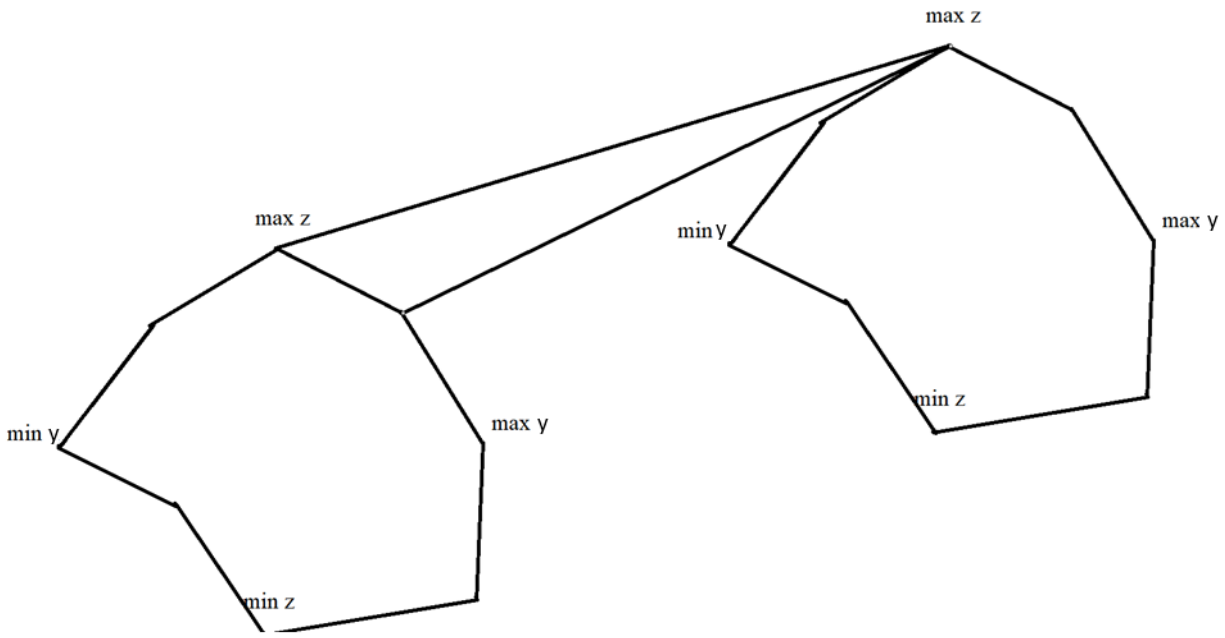


Рисунок 3.4.12. Первый шаг триангуляции стороны для нетривиального случая

Так бы выглядел n-ый(3) шаг триангуляции, если бы в первой группе было бы 4 точки:

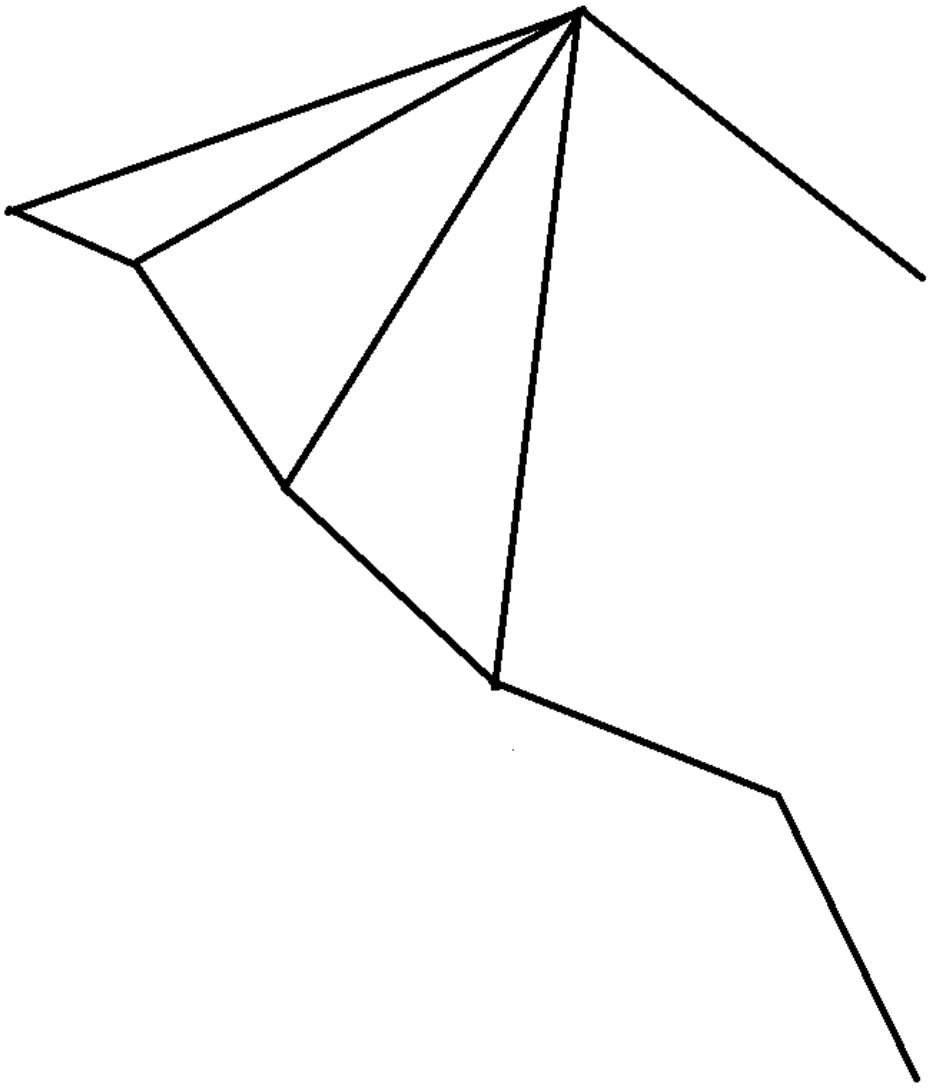


Рисунок 3.4.13. Третий шаг триангуляции для нетривиального случая

Выполняем мы этот алгоритм, пока не дойдём до предпоследней точки (в данном случае мы это уже сделали). Последним шагом мы берем: последнюю точку i группы у первого пикета, первую точку i группы у второго пикета и первую точку i+1 группы у первого пикета.

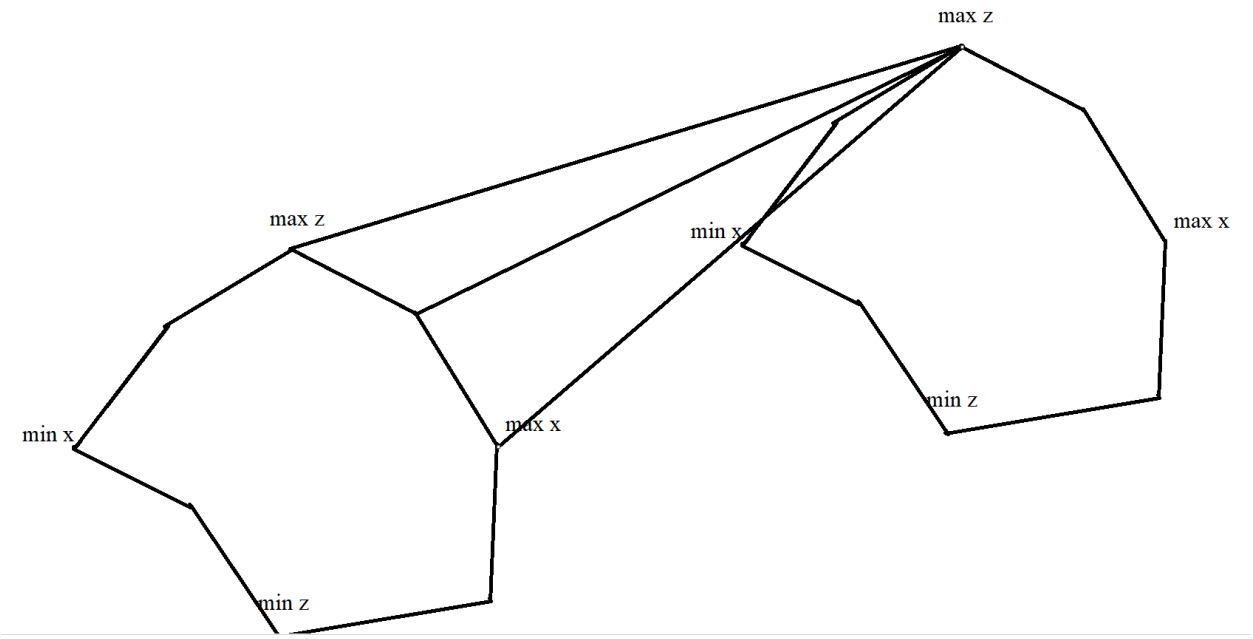


Рисунок 3.4.14. Окончание триангуляции со стороны первого пикета

Теперь проведем триангуляцию со стороны второго пикета. Действуем по тому же алгоритму: берем первую точку i группы у второго пикета, первую точку i+1 группы у первого пикета и вторую точку i группы у второго пикета и передаём в функцию ***Sorted_dots*.**

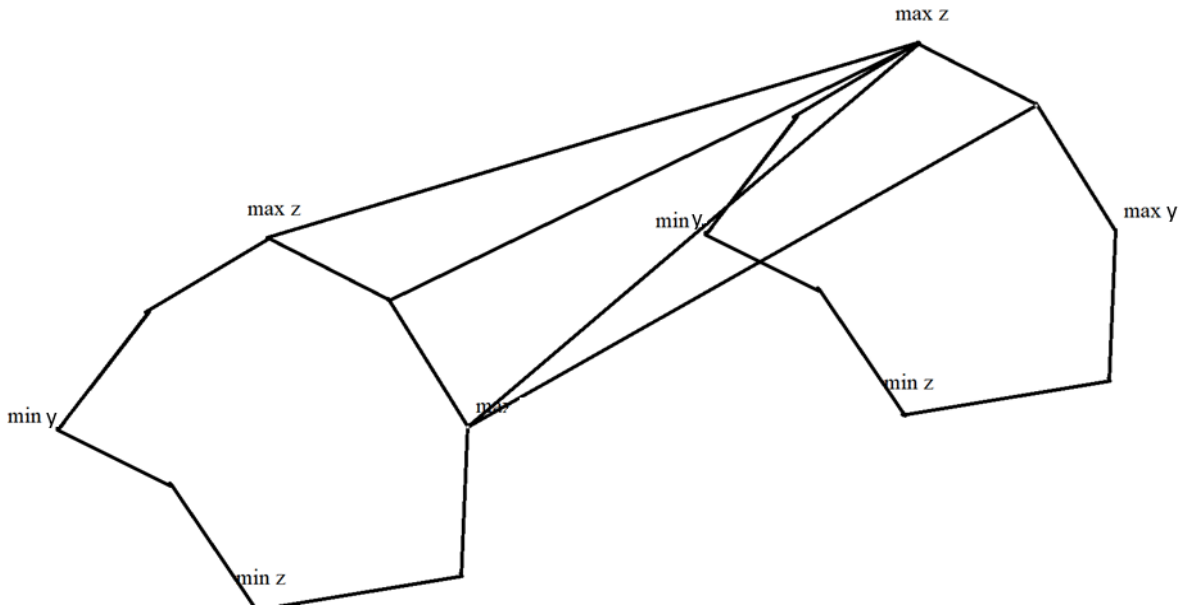


Рисунок 3.4.15. Первый шаг со стороны второго пикета

Вновь выполняем алгоритм пока не дойдем до предпоследней точки. И на последнем шаге: берем последнюю точку i группы у второго пикета, первую

точку i+1 группы у первого пикета и первую точку i+1 группы у второго пикета.

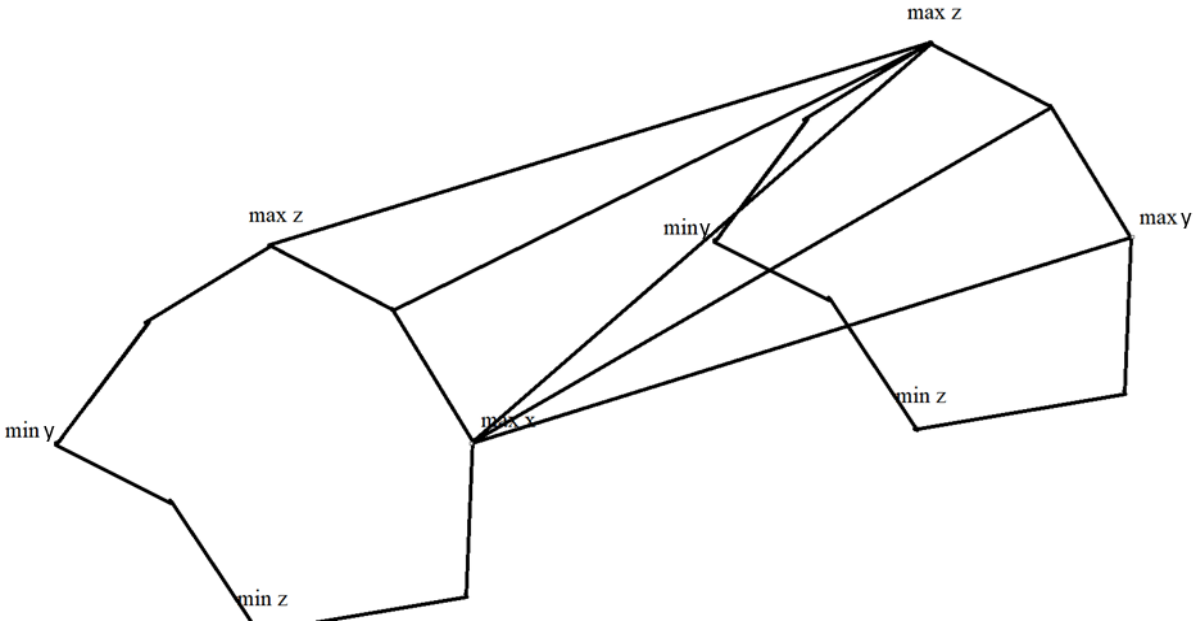


Рисунок 3.4.16. Итог триангуляции одной из сторон

Вот так выглядит триангуляция всех боковых сторон:

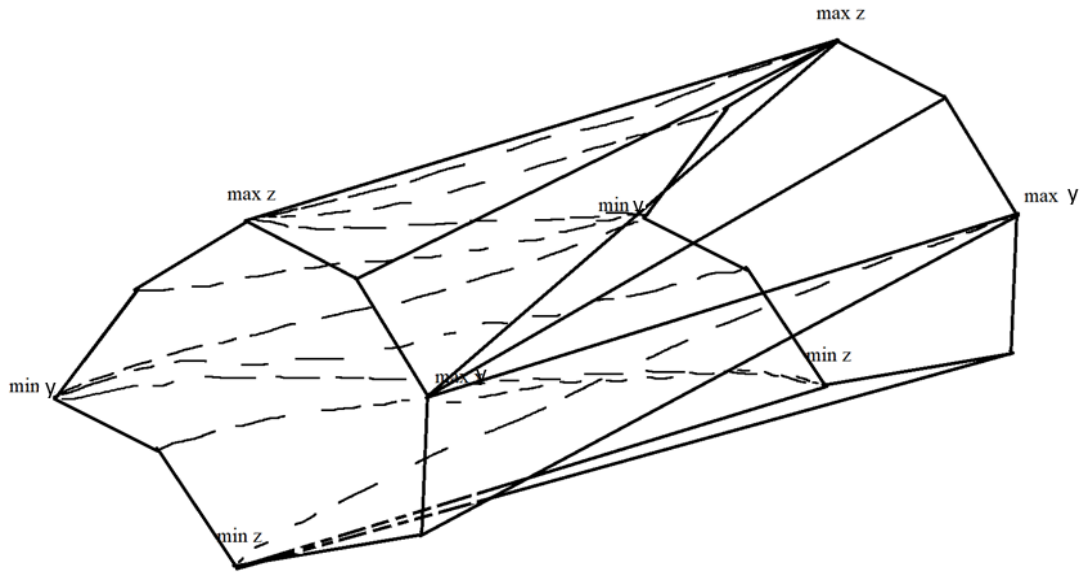


Рисунок 3.4.17. Триангуляции всех боков сторон

Взглянем на триангуляцию оснований. По сравнению с тривиальным случаем существует одно отличие: пока мы выполняем триангуляцию боковых сторон,

мы также проводим триангуляцию и части основания. Мы берем первую точку i группы y первого пикета, вторую точку i группы y первого пикета и первую точку i+1 группы первого пикета передаём в функцию ***Sorted_dots*.** Таким образом мы можем учесть те изгибы основания, которые не охватим при основной триангуляции.

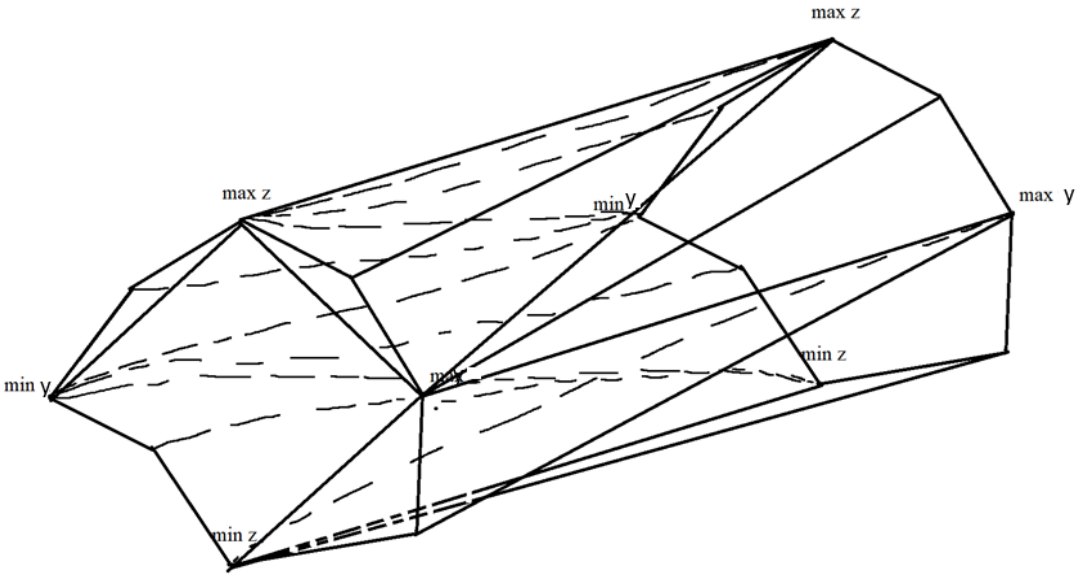


Рисунок 3.4.18. Триангуляции основания

```
                    Numbers = { {},{},{},{} };
                    a = max_elm(Dot[i], 2);
                    b = min_elm(Dot[i], 2);
                    c = max_elm(Dot[i], 1);
                    d = min_elm(Dot[i], 1);
                    dots1 = { {a}, {c}, {b}, {d} };
                    dots1 = dots_original_V(dots1, Dot[i]);
                    a = dots1[0][0]; c = dots1[1][0]; b = dots1[2][0]; d = dots1[3][0];
                    for (int j = 0; j < Dot[i ].size(); j++)
                    {
                        if (Dot[i][j] != a && Dot[i ][j] != b && Dot[i][j] != c &&
Dot[i][j] != d)
                            Numbers[Sort_one(a, b, c, d, Dot[i][j],
lenghtV)].push_back(Dot[i ][j]);
                    }
                    for (int j = 0; j < Numbers[0].size(); j++)
                    {
                        if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
                        {
                            dots1[0].push_back(Numbers[0][j]);
                        }
                        else
                        {
                            dots1[3].push_back(Numbers[0][j]);
                        }
                    }
                    for (int j = 0; j < Numbers[1].size(); j++)
                    {
                        if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
                        {
                            dots1[1].push_back(Numbers[1][j]);
                        }
                        else
                        {
                            dots1[0].push_back(Numbers[1][j]);
                        }
                    }
                    for (int j = 0; j < Numbers[2].size(); j++)
                    {
                        if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
                        {
                            dots1[2].push_back(Numbers[2][j]);
                        }
                        else
                        {
                            dots1[1].push_back(Numbers[2][j]);
                        }
                    }
                    for (int j = 0; j < Numbers[3].size(); j++)
                    {
                        if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
                        {
                            dots1[3].push_back(Numbers[3][j]);
                        }
                        else
                        {
                            dots1[2].push_back(Numbers[3][j]);
                        }
                    }
```

Листинг 3.4.5. Триангуляция

```
            dots1 = Sort_lenghts(dots1, lenghtV);
            dots2 = Sort_lenghts(dots2, lenghtV);


            V1[0] = (dots1[0][0][0] + dots1[1][0][0] + dots1[3][0][0] +
dots1[2][0][0]) * 0.25; V1[1] = (dots1[0][0][1] + dots1[1][0][1] +
dots1[3][0][1] + dots1[2][0][1]) * 0.25; V1[2] = (dots1[0][0][2] +
dots1[1][0][2] + dots1[3][0][2] + dots1[2][0][2]) * 0.25;
            V2[0] = (dots2[0][0][0] + dots2[1][0][0] + dots2[3][0][0] +
dots2[2][0][0]) * 0.25; V2[1] = (dots2[0][0][1] + dots2[1][0][1] +
dots2[3][0][1] + dots2[2][0][1]) * 0.25; V2[2] = (dots2[0][0][2] +
dots2[1][0][2] + dots2[3][0][2] + dots2[2][0][2]) * 0.25;
            for (int j = 0; j < 4; j++)
            {

                if (dots1[j].size() > 1)
                {

                    for (int k = 0; k < dots1[j].size() - 1; k++)
                    {
                        A = dots1[j][k];
                        B = dots2[j][0];
                        C = dots1[j][k+1];




                        V += Sorted_dots(A,B,C, V1, V2);
                        V += Sorted_dots(A, C, dots1[(j + 1) % 4][0], V1,
V2);

                    }
                    A = dots1[j ][dots1[j ].size()-1];
                    B = dots2[j ][0];
                    C = dots1[(j + 1) % 4][0];



                    V += Sorted_dots(A, B, C, V1, V2);
                }
                else
                {
                    A = dots1[j ][0];
                    B = dots2[j ][0];
                    C = dots1[(j + 1) % 4][0];



                    V += Sorted_dots(A, B, C, V1, V2);;
                }
```

```
                if (dots2[j].size() > 1)
                {
                    for (int k = 0; k < dots2[j].size() - 1; k++)
                    {
                        A = dots2[j][k];
                        B = dots1[(j + 1) % 4][0];
                        C = dots2[j][k + 1];



                        V += Sorted_dots(A, B, C, V1, V2);
                        V += Sorted_dots(A, C, dots2[(j + 1) % 4][0], V1,
V2);

                    }

                }

                A = dots2[j][dots2[j].size()-1];
                B = dots1[(j + 1) % 4][0];
                C = dots2[(j + 1) % 4][0];



                V += Sorted_dots(A, B, C, V1, V2);

            }


            V += Sorted_dots(dots1[0][0], dots1[3][0], dots1[1][0], V1,
V2);
            V += Sorted_dots(dots1[2][0], dots1[3][0], dots1[1][0], V1,
V2);

            V += Sorted_dots(dots2[0][0], dots2[3][0], dots2[1][0], V1,
V2);
            V += Sorted_dots(dots2[2][0], dots2[3][0], dots2[1][0], V1,
V2);
```

Листинг 3.4.6. Вычисления объёма

Глава 4. Результаты

Дабы убедится в корректности работы функций было произведено ручное вычисление площади и объёма участков пещеры. Были использованы участки 3 пещер: «Шея», «Галерея дракона», «Зречка».

Для каждого метода будем рассматривать 3 параметра: численное значение, сравнение результата функции с примерным значением и точность метода (точность метода высчитывается сложением сравнений для каждой пары пикетов и делением на количество пар).

Результаты по пещере «Шея»:

Ручное вычисление площади: ≈ 2144 $м^2$

Результат 1 метода: ≈ 2512.53$м^2$

В процентах от примерного значения: 1.17189

Точность метода: 1.07531

Результат 2 метода: ≈ 2373.94$м^2$

В процентах от примерного значения: 1.10725

Точность метода: 1.04103

Результат 3 метода: ≈ 2052.64$м^2$

В процентах от примерного значения: 0.95739

Точность метода: 0.97866

Результат 4 метода: ≈ 2421.92 $м^2$

В процентах от примерного значения: 1.12962

Точность метода: 1.02018

Ручное вычисление объёма: ≈ 33632$м^3$

Результат метода «Эллипсоид»: ≈ 24962.8 $м^3$

В процентах от примерного значения: 0.742232

Точность метода: 0.715899

Результат метода «Эллипсоид» 2 версия: ≈ 32051.4 $м^3$

В процентах от примерного значения: 0.953002

Точность метода: 0.765814

Результат метода «Усеченной пирамиды»: ≈ 30342.6 м^3

В процентах от примерного значения: 0.902194

Точность метода: 0.562645

Результат метода «Усеченной пирамиды» 2 версия: ≈ 38933.1 м^3

В процентах от примерного значения: 1.15762

Точность метода: 0.992694

Результат метода «Полная триангуляция»: ≈ 27392.7 м^3

В процентах от примерного значения: 0.814483

Точность метода: 0.52631

Результат метода «Базовая триангуляция»: ≈ 15685.8м^3

В процентах от примерного значения: 0.466394

Точность метода: 0.40129

Результат метода «Экстраполированная базовая триангуляция»: ≈ 29137.9 м^3

В процентах от примерного значения: 0.866375

Точность метода: 0.568563

Результаты по пещере «Зречка»:

Ручное вычисление площади: ≈ 1258 м^2

Результат 1 метода: ≈ 1028.12 м^2

В процентах от примерного значения: 0.817266

Точность метода: 1.06695

Результат 2 метода: ≈ 1032.77 м^2

В процентах от примерного значения: 0.820959

Точность метода: 0.910988

Результат 3 метода: ≈ 1016.01 м^2

В процентах от примерного значения: 0.807637

Точность метода: 0.862594

Результат 4 метода: ≈ 1042.18 м^2

В процентах от примерного значения: 0.828441

Точность метода: 0.858767

Ручное вычисление объёма: ≈ 9051.12м^3

Результат метода «Эллипсоид»: ≈ 9724.17 м^3

В процентах от примерного значения: 1.07436

Точность метода: 0.994484

Результат метода «Эллипсоид» 2 версия: ≈ 8742.93м^3

В процентах от примерного значения: 0.96595

Точность метода: 1.03235

Результат метода «Усеченной пирамиды»: ≈ 5044.58м^3

В процентах от примерного значения: 0.557344

Точность метода: 0.844518

Результат метода «Усеченной пирамиды» 2 версия: ≈ 13742.7 м^3

В процентах от примерного значения: 1.51834

Точность метода: 1.19769

Результат метода «Полная триангуляция»: ≈ 6064.76 м^3

В процентах от примерного значения: 0.670057

Точность метода: 0.684939

Результат метода «Базовая триангуляция»: ≈ 5343.67 м^3

В процентах от примерного значения: 0.590388

Точность метода: 0.576776

Результат метода «Экстраполированная базовая триангуляция»: ≈ 5033.74 м^3

В процентах от примерного значения: 0.556145

Точность метода: 0.821339

Результаты по пещере «Галерея дракона»:

Ручное вычисление площади: ≈ 406м^2

Результат 1 метода: ≈ 169.248 м^2

В процентах от примерного значения: 0.416867

Точность метода: 0.766441

Результат 2 метода: ≈ 252.325 м^2

В процентах от примерного значения: 0.62149

Точность метода: 0.998943

Результат 3 метода: ≈ 229.184м^2

В процентах от примерного значения: 0.564492

Точность метода: 0.832681

Результат 4 метода: ≈ 244.888 м^2

В процентах от примерного значения: 0.603173

Точность метода: 0.901257

Ручное вычисление объёма: ≈ 2008.5 м^3

Результат метода «Эллипсоид»: ≈ 1221.18 м^3

В процентах от примерного значения: 0.608008

Точность метода: 0.707326

Результат метода «Эллипсоид» 2 версия: ≈ 846.007 м^3

В процентах от примерного значения: 0.421213

Точность метода: 0.837615

Результат метода «Усеченной пирамиды»: ≈ 647.603 м^3

В процентах от примерного значения: 0.322431

Точность метода: 0.485925

Результат метода «Усеченной пирамиды» 2 версия: ≈ 2359.19 м^3

В процентах от примерного значения: 1.1746

Точность метода: 1.47855

Результат метода «Полная триангуляция»: ≈ 1592.86 м^3

В процентах от примерного значения: 0.79306

Точность метода: 0.816692

Результат метода «Базовая триангуляция»: ≈ 802.614 м^3

В процентах от примерного значения: 0.399609

Точность метода: 0.428004

Результат метода «Экстраполированная базовая триангуляция»: ≈ 517.847 м^3

В процентах от примерного значения: 0.257828

Точность метода: 0.46302

Заключение

Было разработано несколько методов для вычисления площади пола и объёма пещеры для случая, когда форма пещеры описывалась с помощью отстрелов. Они показали различные результаты.

Метод «Базовой триангуляции» полностью не справился с поставленной задачей. То, что результат при использовании данного метода будет меньше, было ожидаемо, но то, насколько выдаваемый результат оказался меньше, делает использование этого метода бессмысленным.

Также стоит сказать о работе функций на пещере «Галерея Дракона». На этом массиве данных точность практически всех функций значительно упала (в некоторых случаях выдаваемы значения сильно не достигают необходимой точности). При этом функции выдают отличную точность при работе с отдельными участками. Данный результат показывает необходимость дальнейшей проверки работоспособности дабы выявить другие особые случаи.

При поиске площади наилучший результат показал 2 и 4 метод. 3 метод, напротив, имеет большую погрешность чем 4 метод, но всё ещё в пределах допустимого. В будущем необходимо будет провести более тщательный анализ и определить, является ли подобный результат недостатком реализации или для решения поставленной задачи 4 метод является более подходящим.

Среди функций, используемых для поиска объёма, та, в которой реализован второй вариант метода «Эллипсоид» показала наилучший результат. Среди методов один и два 2 вариант функций выдает более точные результаты. Стоит провести более тщательный анализ функций метода 2, так как несмотря на то, что на первом массиве данных результат является достаточно хорошим, на втором точность близка к пересечению порога нормы. Нужно определить, чем вызвано падение точности. Метод «Полной триангуляции» в целом показал среднюю точность, но при этом разброс точности на отдельных парах пикетов достаточно высок. Необходимо определить причину этого. Метод «Экстраполированной базовой триангуляции» показывает невысокую точность, но всё ещё в пределах допустимого. Благодаря своей простате вполне вероятно, что данный метод может быть использован, даже не смотря на низкую точность.

В целом, необходимо провести больше тестов на различных пещерах.

Одно из возможных направлений развития – реализация метода, осуществляющего поиск объёма с помощью ориентированных объёмов (близкий к методу ориентированных площадей). Другой путь – триангуляция с использованием сплайнов.

Список литературы

1. Серафимов К.Б. Цифровая топосъемка в пещере: методика для соло с «оркестром». 2016.
2. Дегятрев А. П. Теория и практика спелеотопосъемки. 2001.
3. Грачев А. П. Топографо-геодезические работы в горизонтальных пещерах. Практические рекомендации для спелеотопографа. – К, 2010. – 48 с.
4. Жихарев С.А., Скворцов А.В. Моделирование рельефа в системе ГрафИн. Геоинформатика: Теория и практика. Издательство Томского университета. 1998
5. Лопшиц А. М. Вычисление площадей ориентированных фигур. — М. : Гостехиздат, 1956. — 60 с. — (Популярные лекции по математике ; вып. 20).

Приложение

```
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <vector>
#include<limits>
#include <fstream>
using namespace std;

template<typename T>
T lenght(vector<T> a, vector<T> b)
{
    return sqrt(((b[0] - a[0]) * (b[0] - a[0]) + (b[1] - a[1]) * (b[1] - a[1]) + (b[2] - a[2]) * (b[2] - a[2])));
}

template<typename T>
T lenghtV(vector<T> a, vector<T> b)
{
    return sqrt(((b[1] - a[1]) * (b[1] - a[1]) + (b[2] - a[2]) * (b[2] - a[2])));
}

template<typename T>
T lenghtS(vector<T> a, vector<T> b)
{
    return sqrt(((b[0] - a[0]) * (b[0] - a[0]) + (b[1] - a[1]) * (b[1] - a[1])));
}
```

```
template<typename T>
vector < vector <vector<T>>> Dots_interpenter(vector < vector <vector<T>>>& picket)
{
    vector < vector < vector <T> > > Dot;
    vector < vector <T> > Tdot;
    vector<T> dot{ 0, 0, 0 };
    vector<T> dot1{ 0, 0, 0 };
    vector<T> dot2{ 0, 0, 0 };
    T in_gr = M_PI / 180;



    for (int i = 0; i < size(picket); i++)
    {
        dot2[0] +=  picket[i][0][0] * cos(picket[i][0][1] * in_gr) * cos(picket[i][0][2] * in_gr);
        dot2[1] +=  picket[i][0][0] * sin(picket[i][0][1] * in_gr) * cos(picket[i][0][2] * in_gr);
        dot2[2] +=  picket[i][0][0] * sin((picket[i][0][2] * in_gr));



        for (int j = 1; j < size(picket[i]); j++)
        {
            dot[0] = dot2[0] + picket[i][j][0] * cos(picket[i][j][1] * in_gr) * cos(picket[i][j][2] * in_gr);
            dot[1] = dot2[1] + picket[i][j][0] * sin(picket[i][j][1] * in_gr) * cos(picket[i][j][2] * in_gr);
            dot[2] = dot2[2] + picket[i][j][0] * sin((picket[i][j][2] * in_gr));
```

```
            Tdot.push_back(dot);
        }
        Dot.push_back(Tdot);
        Tdot.clear();
        dot1 = dot2;
    }



    return Dot;
}



template<typename T>
T area_of_triangle_Ger(vector<T> dot1, vector<T> dot2, vector<T> dot3)
{
    T a, b, c, p;
    a = sqrt((dot1[0] - dot2[0]) * (dot1[0] - dot2[0]) + (dot1[1] - dot2[1]) * (dot1[1] - dot2[1]));


    b = sqrt((dot2[0] - dot3[0]) * (dot2[0] - dot3[0]) + (dot2[1] - dot3[1]) * (dot2[1] - dot3[1]));


    c = sqrt((dot1[0] - dot3[0]) * (dot1[0] - dot3[0]) + (dot1[1] - dot3[1]) * (dot1[1] - dot3[1])
);


    p = (a + b + c) * 0.5;
    double d = sqrt(p * (p - a) * (p - b) * (p - c));
```

```
    return sqrt(p * (p - a) * (p - b) * (p - c));
}

template<typename T>
vector<T> max_elm(vector <vector<T>> dots, int j)
{
    int numb = 0;
    T max = dots[0][j];
    for (int i = 1; i < dots.size(); i++)
    {
        if (dots[i][j] > max)
        {
            numb = i;
            max = dots[i][j];
        }
    }
    return dots[numb];
}

template<typename T>
vector<T> max_elm_exp(vector <vector<T>> dots,  vector<T> a, int j)
{
    int numb = 0;
    T max;
    if(dots[0] != a)
        max = dots[0][j];
    else
        max = dots[1][j];
```

```
    for (int i = 1; i < dots.size(); i++)
    {
        if (dots[i][j] > max && dots[i] != a)
        {
            numb = i;
            max = dots[i][j];
        }
    }
    return dots[numb];
}

template<typename T>
vector<T> min_elm(vector <vector<T>> dots, int j)
{
    int numb = 0;
    T min = dots[0][j];
    for (int i = 1; i < dots.size(); i++)
    {
        if (dots[i][j] <  min)
        {
            numb = i;
            min = dots[i][j];
        }
    }
    return dots[numb];
}
```

```
template<typename T>
vector<T> min_elm_exp( vector <vector<T>> dots, vector<T> a, int j)
{
   int numb = 0;
   T min;

   if (dots[0] != a)
      min = dots[0][j];
   else
      min = dots[1][j];

   for (int i = 1; i < dots.size(); i++)
   {
      if (dots[i][j] < min && dots[i] != a)
      {
         numb = i;
         min = dots[i][j];
      }
   }
   return dots[numb];
}




template<typename T>
T Cave_S(vector < vector <vector<T>>> Dot)
{
```

```
T S = 0;
vector<T> a, b,c,d,e,f;

for (int i = 0; i < Dot.size() - 2; i++)
{
    a = max_elm(Dot[i], 1);
    b = min_elm(Dot[i], 1);
    c = max_elm(Dot[i+1], 1);
    d = min_elm(Dot[i+1], 1);
    e = min_elm(Dot[i], 0);
    f = min_elm(Dot[i + 1], 0);
    a[0] = b[0] = e[0];
    c[0] = d[0] = f[0];
    S += area_of_triangle_Ger(b, a, d);
    S += area_of_triangle_Ger(d,c, a);

}
a = max_elm(Dot[Dot.size() - 2], 1);
b = min_elm(Dot[Dot.size() - 2], 1);
c = max_elm(Dot[Dot.size() - 1], 1);
d = min_elm(Dot[Dot.size() - 1], 1);
e = min_elm(Dot[Dot.size() - 2], 0);
f = max_elm(Dot[Dot.size() - 1], 0);
a[0] = b[0] = e[0];
c[0] = d[0] = f[0];
S += area_of_triangle_Ger(b, a, d);
S += area_of_triangle_Ger(d, c, a);
```

```
    return S;
}

template<typename T>
 vector <T>  V_Zero(vector<T> Zero, vector <T>  dots)
{

    for (int k = 0; k < 3; k++)
    {
        dots[k] =   dots[k] - Zero[k];
    }

    return dots;
}

template<typename T>
T Determinant(vector<T> a, vector<T> b, vector<T> c)
{
    T l = abs(a[0] * (b[1] * c[2] - b[2] * c[1]) - b[0] * (a[1] * c[2] - a[2] * c[1]) +
        c[0] * (a[1] * b[2] - a[2] * b[1]));
    return abs(a[0] * (b[1] * c[2] - b[2] * c[1]) - b[0] * (a[1] * c[2] - a[2] * c[1]) +
        c[0] * (a[1] * b[2] - a[2] * b[1]));
}



template<typename T>
bool AreSame(T a, T b)
```

```
{
    return fabs(a - b) < 0.0000001;
}

template<typename T>
T Sonor(vector<T> A, vector<T> B)
{
    T temp1, temp2, temp3;
    temp1 = A[0] * B[0] + A[1] * B[1] + A[2] * B[2];
    temp2 = sqrt(A[0] * A[0] + A[1] * A[1] + A[2] * A[2]);
    temp3 = sqrt(B[0] * B[0] + B[1] * B[1] + B[2] * B[2]);
    temp1 = temp1 / (temp2 * temp3);
    if(AreSame(temp1,1.0))
        temp1 = 1.0;
    else if(AreSame(temp1, -1.0))
        temp1 = -1.0;
    return acos(temp1);

}

template<typename T>
vector<T> Vector_P(vector<T> A, vector<T> B)
{
    vector<T> Ans{ 0, 0, 0 };
    Ans[0] = A[1] * B[2] - A[2] * B[1];
    Ans[1] = -(A[0] * B[2] - A[2] * B[0]);
    Ans[2] = A[0] * B[1] - A[1] * B[0];
    return Ans;
```

```
}


template<typename T>
T  Sorted_dots(vector<T> A, vector<T> B, vector<T> C, vector<T> Cen1, vector<T> Cen2)
{
   vector<T> Zero{ 0, 0, 0 };
   vector<T> T1{ 0, 0, 0 };
   vector<T> T2{ 0, 0, 0 };
   vector<T> T3{ 0, 0, 0 };
   Zero[0] = (Cen1[0] + Cen2[0]) * 0.5;
   Zero[1] = (Cen1[1] + Cen2[1]) * 0.5;
   Zero[2] = (Cen1[2] + Cen2[2]) * 0.5;

   T1[0] = A[0] - Zero[0]; T1[1] = A[1] - Zero[1]; T1[2] = A[2] - Zero[2];
   T2[0] = B[0] - Zero[0]; T2[1] = B[1] - Zero[1]; T2[2] = B[2] - Zero[2];
   T3[0] = C[0] - Zero[0]; T3[1] = C[1] - Zero[1]; T3[2] = C[2] - Zero[2];
   //T1[0] = Zero[0] - A[0]; T1[1] = Zero[1] - A[1]; T1[2] = Zero[2] - A[2];
   //T2[0] = B[0] - A[0]; T2[1] = B[1] - A[1]; T2[2] = B[2] - A[2];
   //T3[0] = C[0] - A[0]; T3[1] = C[1] - A[1]; T3[2] = C[2] - A[2];

   return Determinant(T1, T2, T3);


}

template<typename T>
```

```
vector < vector < vector <T> > > Sort_lenghts(vector < vector < vector <T> > > dots , T(*lenght)(vector<T>, vector<T>))
{
    vector <T>  a;
    int n = 0;
    for (int i = 0; i < dots.size(); i++)
    {
        a = dots[i][0];
        for (int j = 1; j < dots[i].size(); j++)
        {

            for (int k = 1; k < dots[i].size()-1; k++)
            {
                if (lenght(dots[i][k], a) > lenght(dots[i][k+1], a))
                {
                    swap(dots[i][k], dots[i][k+1]);
                }
            }
        }

    }
    return dots;
}

template<typename T>
int Sort_one(vector<T> a, vector<T> b, vector<T> c, vector<T> d , vector<T> dot , T(*lenght)(vector<T>, vector<T>))
{
    T S_lenght = lenght(a, dot);
```

```
    int number = 0;
    if (lenght(c, dot) < S_lenght)
    {
        S_lenght = lenght(c, dot);
        number = 1;
    }
    if (lenght(b, dot) < S_lenght)
    {
        S_lenght = lenght(b, dot);
        number = 2;
    }
    if (lenght(d, dot) < S_lenght)
    {
        S_lenght = lenght(d, dot);
        number = 3;
    }
    return number;
}

template<typename T>
vector < vector < vector <T> > > dots_original_V(vector < vector < vector <T> > > dots, vector < vector <T> >  Dot)
{
    if (dots[0][0] == dots[3][0])
        dots[3][0] = min_elm_exp(Dot, dots[0][0], 1);

    if (dots[0][0] == dots[1][0])
        dots[1][0] = max_elm_exp(Dot, dots[0][0], 1);
```

```
    if (dots[2][0] == dots[3][0])
        dots[3][0] = min_elm_exp(Dot, dots[2][0], 1);

    if (dots[2][0] == dots[1][0])
        dots[1][0] = max_elm_exp(Dot, dots[2][0], 1);

    return dots;
}

template<typename T>
T Cave_V(vector < vector <vector<T>>> Dot)
{
    T V = 0;

    vector<T> a, b, c, d;
    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector<T> V1{ 0, 0, 0 }, V2{ 0, 0, 0 };
    vector<T> A{ 0, 0, 0 };
    vector<T> B{ 0, 0, 0 };
    vector<T> C{ 0, 0, 0 };
    vector < vector < vector <T> >> Numbers;
    for (int i = 0; i < Dot.size() - 1; i++)
    {

        if (Dot[i].size() == 4 && Dot[i+1].size() == 4)
        {
```

```
        dots1 = { {max_elm(Dot[i], 2)}, {max_elm(Dot[i], 1)}, {min_elm(Dot[i], 2)}, {min_elm(Dot[i], 1)} };

        dots1 = dots_original_V(dots1, Dot[i]);

        dots2 = { {max_elm(Dot[i + 1], 2)}, {max_elm(Dot[i + 1], 1)} , {min_elm(Dot[i + 1], 2)}, {min_elm(Dot[i + 1], 1)} };

        dots2 = dots_original_V(dots2, Dot[i+1]);


        for (int j = 0; j < 4; j++)

        {


            V1[0] = (dots1[0][0][0] + dots1[1][0][0] + dots1[3][0][0] + dots1[2][0][0]) * 0.25; V1[1] = (dots1[0][0][1] + dots1[1][0][1] + dots1[3][0][1] + dots1[2][0][1]) * 0.25; V1[2] = (dots1[0][0][2] + dots1[1][0][2] + dots1[3][0][2] + dots1[2][0][2]) * 0.25;

            V2[0] = (dots2[0][0][0] + dots2[1][0][0] + dots2[3][0][0] + dots2[2][0][0]) * 0.25; V2[1] = (dots2[0][0][1] + dots2[1][0][1] + dots2[3][0][1] + dots2[2][0][1]) * 0.25; V2[2] = (dots2[0][0][2] + dots2[1][0][2] + dots2[3][0][2] + dots2[2][0][2]) * 0.25;


            A = dots1[j][0];

            B = dots2[j][0];

            C = dots1[(j + 1) % 4][0];

            V += Sorted_dots(A, B, C, V1, V2);




            A = dots2[j][0];

            B = dots1[(j + 1) % 4][0];

            C = dots2[(j + 1) % 4][0];

            V += Sorted_dots(A, C, B, V1, V2);

        }
```

```
            V += Sorted_dots(dots1[0][0], dots1[3][0], dots1[1][0], V1, V2);
            V += Sorted_dots(dots1[2][0], dots1[3][0], dots1[1][0], V1, V2);

            V += Sorted_dots(dots2[0][0], dots2[3][0], dots2[1][0], V1, V2);
            V += Sorted_dots(dots2[2][0], dots2[3][0], dots2[1][0], V1, V2);

        }
        else
        {
            if (Dot[i].size() > 4)
            {
                Numbers = { {},{},{},{} };
                a = max_elm(Dot[i], 2);
                b = min_elm(Dot[i], 2);
                c = max_elm(Dot[i], 1);
                d = min_elm(Dot[i], 1);
                dots1 = { {a}, {c}, {b}, {d} };
                dots1 = dots_original_V(dots1, Dot[i]);
                a = dots1[0][0]; c = dots1[1][0]; b = dots1[2][0]; d = dots1[3][0];
                for (int j = 0; j < Dot[i ].size(); j++)
                {
                    if (Dot[i][j] != a && Dot[i ][j] != b && Dot[i][j] != c && Dot[i][j] !=
d)
                        Numbers[Sort_one(a, b, c, d, Dot[i][j], lenghtV)].push_back(Dot[i
][j]);

                }
```

```
for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
   {
      dots1[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots1[3].push_back(Numbers[0][j]);
   }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
   {
      dots1[1].push_back(Numbers[1][j]);
   }
   else
   {
      dots1[0].push_back(Numbers[1][j]);
   }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
   if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
   {
```

```
                dots1[2].push_back(Numbers[2][j]);
            }
            else
            {
                dots1[1].push_back(Numbers[2][j]);
            }
        }

        for (int j = 0; j < Numbers[3].size(); j++)
        {
            if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
            {
                dots1[3].push_back(Numbers[3][j]);
            }
            else
            {
                dots1[2].push_back(Numbers[3][j]);
            }
        }

    }
    else
    {
        dots1 = { {max_elm(Dot[i], 2)}, {max_elm(Dot[i], 1)},
{min_elm(Dot[i], 2)}, {min_elm(Dot[i], 1)} };
        dots1 = dots_original_V(dots1, Dot[i]);
    }
```

```
if (Dot[i+1].size() > 4)
{
   Numbers = { {},{},{},{} };
   a = max_elm(Dot[i + 1], 2);
   b = min_elm(Dot[i + 1], 2);
   c = max_elm(Dot[i + 1], 1);
   d = min_elm(Dot[i + 1], 1);
   dots2 = { {a}, {c}, {b}, {d} };
   dots2 = dots_original_V(dots2, Dot[i + 1]);
   a = dots2[0][0]; c = dots2[1][0]; b = dots2[2][0]; d = dots2[3][0];
   for (int j = 0; j < Dot[i + 1].size(); j++)
   {
      if(Dot[i + 1][j] !=a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
         Numbers[Sort_one(a,b,c,d, Dot[i + 1][j], lenghtV)].push_back(Dot[i
+ 1][j]);


   }


   for (int j = 0; j < Numbers[0].size(); j++)
   {
      if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
      {
         dots2[0].push_back(Numbers[0][j]);
      }
      else
```

```
    {
        dots2[3].push_back(Numbers[0][j]);
    }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
    if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
    {
        dots2[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots2[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots2[1].push_back(Numbers[2][j]);
    }
}
```

```
            for (int j = 0; j < Numbers[3].size(); j++)
            {
                if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
                {
                    dots2[3].push_back(Numbers[3][j]);
                }
                else
                {
                    dots2[2].push_back(Numbers[3][j]);
                }
            }
            Numbers.clear();


        }
        else
        {
            dots2 = { {max_elm(Dot[i + 1], 2)}, {max_elm(Dot[i + 1], 1)} ,
{min_elm(Dot[i + 1], 2)}, {min_elm(Dot[i + 1], 1)} };
            dots2 = dots_original_V(dots2, Dot[i + 1]);
        }


        dots1 = Sort_lenghts(dots1, lenghtV);
        dots2 = Sort_lenghts(dots2, lenghtV);



        V1[0] = (dots1[0][0][0] + dots1[1][0][0] + dots1[3][0][0] + dots1[2][0][0])
* 0.25; V1[1] = (dots1[0][0][1] + dots1[1][0][1] + dots1[3][0][1] + dots1[2][0][1])
```

```
* 0.25; V1[2] = (dots1[0][0][2] + dots1[1][0][2] + dots1[3][0][2] + dots1[2][0][2])
* 0.25;

        V2[0] = (dots2[0][0][0] + dots2[1][0][0] + dots2[3][0][0] + dots2[2][0][0])
* 0.25; V2[1] = (dots2[0][0][1] + dots2[1][0][1] + dots2[3][0][1] + dots2[2][0][1])
* 0.25; V2[2] = (dots2[0][0][2] + dots2[1][0][2] + dots2[3][0][2] + dots2[2][0][2])
* 0.25;

        for (int j = 0; j < 4; j++)

        {


           if (dots1[j].size() > 1)

           {


              for (int k = 0; k < dots1[j].size() - 1; k++)

              {

                 A = dots1[j][k];

                 B = dots2[j][0];

                 C = dots1[j][k+1];









                 V += Sorted_dots(A,B,C, V1, V2);

                 V += Sorted_dots(A, C, dots1[(j + 1) % 4][0], V1, V2);


              }

              A = dots1[j ][dots1[j ].size()-1];

              B = dots2[j ][0];

              C = dots1[(j + 1) % 4][0];
```

```
    V += Sorted_dots(A, B, C, V1, V2);
}
else
{
    A = dots1[j ][0];
    B = dots2[j ][0];
    C = dots1[(j + 1) % 4][0];




    V += Sorted_dots(A, B, C, V1, V2);;
}

if (dots2[j].size() > 1)
{
    for (int k = 0; k < dots2[j].size() - 1; k++)
    {
        A = dots2[j][k];
        B = dots1[(j + 1) % 4][0];
        C = dots2[j][k + 1];
```

```
            V += Sorted_dots(A, B, C, V1, V2);
            V += Sorted_dots(A, C, dots2[(j + 1) % 4][0], V1, V2);

        }

    }

    A = dots2[j][dots2[j].size()-1];
    B = dots1[(j + 1) % 4][0];
    C = dots2[(j + 1) % 4][0];




    V += Sorted_dots(A, B, C, V1, V2);

  }


  V += Sorted_dots(dots1[0][0], dots1[3][0], dots1[1][0], V1, V2);
  V += Sorted_dots(dots1[2][0], dots1[3][0], dots1[1][0], V1, V2);

  V += Sorted_dots(dots2[0][0], dots2[3][0], dots2[1][0], V1, V2);
  V += Sorted_dots(dots2[2][0], dots2[3][0], dots2[1][0], V1, V2);
}
dots1.clear();
dots2.clear();
```

```
    }
    return V/6;
}

template<typename T>
T Cave_V_BN1(vector < vector <vector<T>>> Dot)
{
    T V = 0;

    vector<T> a, b, c, d;
    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector<T> V1{ 0, 0, 0 }, V2{ 0, 0, 0 };
    vector<T> A{ 0, 0, 0 };
    vector<T> B{ 0, 0, 0 };
    vector<T> C{ 0, 0, 0 };
    vector < vector < vector <T> >> Numbers;
    for (int i = 0; i < Dot.size() - 1; i++)
    {
        dots1 = { {max_elm(Dot[i], 2)}, {max_elm(Dot[i], 1)}, {min_elm(Dot[i], 2)}, {min_elm(Dot[i], 1)} };
        dots1 = dots_original_V(dots1, Dot[i]);
        dots2 = { {max_elm(Dot[i + 1], 2)}, {max_elm(Dot[i + 1], 1)} , {min_elm(Dot[i + 1], 2)}, {min_elm(Dot[i + 1], 1)} };
        dots2 = dots_original_V(dots2, Dot[i + 1]);

        for (int j = 0; j < 4; j++)
        {
```

```
            V1[0] = (dots1[0][0][0] + dots1[2][0][0]) * 0.5; V1[1] = (dots1[0][0][1]
+ dots1[2][0][1]) * 0.5; V1[2] = (dots1[0][0][2] + dots1[2][0][2]) * 0.5;

            V2[0] = (dots2[0][0][0] + dots2[2][0][0]) * 0.5; V2[1] = (dots2[0][0][1]
+ dots2[2][0][1]) * 0.5; V2[2] = (dots2[0][0][2] + dots2[2][0][2]) * 0.5;


               A = dots1[j][0];

               B = dots2[j][0];

               C = dots1[(j + 1) % 4][0];

               V += Sorted_dots(A, B, C, V1, V2);




               A = dots2[j][0];

               B = dots1[(j + 1) % 4][0];

               C = dots2[(j + 1) % 4][0];

               V += Sorted_dots(A, C, B, V1, V2);

            }


            V += Sorted_dots(dots1[0][0], dots1[3][0], dots1[1][0], V1, V2);

            V += Sorted_dots(dots1[2][0], dots1[3][0], dots1[1][0], V1, V2);


            V += Sorted_dots(dots2[0][0], dots2[3][0], dots2[1][0], V1, V2);

            V += Sorted_dots(dots2[2][0], dots2[3][0], dots2[1][0], V1, V2);




      }


      return V / 6;
```

```
}

template<typename T>
T Cave_V_BN2(vector < vector <vector<T>>> Dot)
{
   T V = 0;

   vector<T> a, b, c, d, e1,e2;
   vector < vector < vector <T> > > dots1;
   vector < vector < vector <T> > > dots2;
   vector<T> V1{ 0, 0, 0 }, V2{ 0, 0, 0 };
   vector<T> A{ 0, 0, 0 };
   vector<T> B{ 0, 0, 0 };
   vector<T> C{ 0, 0, 0 };
   vector < vector < vector <T> >> Numbers;
   for (int i = 0; i < Dot.size() - 2; i++)
   {
      e1 = min_elm(Dot[i], 0);
      e2 = min_elm(Dot[i+1], 0);
      dots1 = { {max_elm(Dot[i], 2)}, {max_elm(Dot[i], 1)}, {min_elm(Dot[i], 2)}, {min_elm(Dot[i], 1)} };
      dots1 = dots_original_V(dots1, Dot[i]);
      dots2 = { {max_elm(Dot[i + 1], 2)}, {max_elm(Dot[i + 1], 1)} , {min_elm(Dot[i + 1], 2)}, {min_elm(Dot[i + 1], 1)} };
      dots2 = dots_original_V(dots2, Dot[i + 1]);
      for (int j = 0; j < 4; j++)
      {
         dots1[j][0][0] = e1[0];
         dots2[j][0][0] = e2[0];
```

```
}

for (int j = 0; j < 4; j++)
{

    V1[0] = (dots1[0][0][0] + dots1[2][0][0]) * 0.5; V1[1] = (dots1[0][0][1] + dots1[2][0][1]) * 0.5; V1[2] = (dots1[0][0][2] + dots1[2][0][2]) * 0.5;
    V2[0] = (dots2[0][0][0] + dots2[2][0][0]) * 0.5; V2[1] = (dots2[0][0][1] + dots2[2][0][1]) * 0.5; V2[2] = (dots2[0][0][2] + dots2[2][0][2]) * 0.5;

    A = dots1[j][0];
    B = dots2[j][0];
    C = dots1[(j + 1) % 4][0];
    V += Sorted_dots(A, B, C, V1, V2);

    A = dots2[j][0];
    B = dots1[(j + 1) % 4][0];
    C = dots2[(j + 1) % 4][0];
    V += Sorted_dots(A, C, B, V1, V2);
}

V += Sorted_dots(dots1[0][0], dots1[3][0], dots1[1][0], V1, V2);
V += Sorted_dots(dots1[2][0], dots1[3][0], dots1[1][0], V1, V2);

V += Sorted_dots(dots2[0][0], dots2[3][0], dots2[1][0], V1, V2);
V += Sorted_dots(dots2[2][0], dots2[3][0], dots2[1][0], V1, V2);
```

```
    }


    e1 = min_elm(Dot[Dot.size() - 2], 0);
    e2 = max_elm(Dot[Dot.size() - 2 + 1], 0);
    dots1 = { {max_elm(Dot[Dot.size() - 2], 2)}, {max_elm(Dot[Dot.size() - 2], 1)}, {min_elm(Dot[Dot.size() - 2], 2)}, {min_elm(Dot[Dot.size() - 2], 1)} };
    dots1 = dots_original_V(dots1, Dot[Dot.size() - 2]);
    dots2 = { {max_elm(Dot[Dot.size() - 2 + 1], 2)}, {max_elm(Dot[Dot.size() - 2 + 1], 1)} , {min_elm(Dot[Dot.size() - 2 + 1], 2)}, {min_elm(Dot[Dot.size() - 2 + 1], 1)} };
    dots2 = dots_original_V(dots2, Dot[Dot.size() - 2 + 1]);


    for (int j = 0; j < 4; j++)
    {
        dots1[j][0][0] = e1[0];
        dots2[j][0][0] = e2[0];
    }


    for (int j = 0; j < 4; j++)
    {


        V1[0] = (dots1[0][0][0] + dots1[2][0][0]) * 0.5; V1[1] = (dots1[0][0][1] + dots1[2][0][1]) * 0.5; V1[2] = (dots1[0][0][2] + dots1[2][0][2]) * 0.5;
        V2[0] = (dots2[0][0][0] + dots2[2][0][0]) * 0.5; V2[1] = (dots2[0][0][1] + dots2[2][0][1]) * 0.5; V2[2] = (dots2[0][0][2] + dots2[2][0][2]) * 0.5;


        A = dots1[j][0];
```

```
        B = dots2[j][0];
        C = dots1[(j + 1) % 4][0];
        V += Sorted_dots(A, B, C, V1, V2);



        A = dots2[j][0];
        B = dots1[(j + 1) % 4][0];
        C = dots2[(j + 1) % 4][0];
        V += Sorted_dots(A, C, B, V1, V2);
    }

    V += Sorted_dots(dots1[0][0], dots1[3][0], dots1[1][0], V1, V2);
    V += Sorted_dots(dots1[2][0], dots1[3][0], dots1[1][0], V1, V2);

    V += Sorted_dots(dots2[0][0], dots2[3][0], dots2[1][0], V1, V2);
    V += Sorted_dots(dots2[2][0], dots2[3][0], dots2[1][0], V1, V2);

    return V / 6;
}

template<typename T>
T Cave_S2(vector < vector <vector<T>>> Dot)
{
    T S = 0;
    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector<T> a, b,c,d;
```

```
vector < vector < vector <T> > >  Numbers;
for (int i = 0; i < Dot.size() - 2; i++)
{
   a = max_elm(Dot[i], 0);
   b = min_elm(Dot[i], 0);
   c = max_elm(Dot[i], 1);
   d = min_elm(Dot[i], 1);
   dots1 = { {a},{c}, {b}, {d} };
   if (Dot[i].size() > 4)
   {
      Numbers = { {},{},{},{} };

      dots1 = { {a}, {c}, {b}, {d} };
      for (int j = 0; j < Dot[i].size(); j++)
      {
         if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c && Dot[i][j] != d)
            Numbers[Sort_one(a, b, c, d, Dot[i][j], lenghtS)].push_back(Dot[i][j]);

      }

      for (int j = 0; j < Numbers[0].size(); j++)
      {
         if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
         {
            dots1[3].push_back(Numbers[0][j]);
         }

      }
```

```
for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
   {
      dots1[1].push_back(Numbers[1][j]);
   }

}

for (int j = 0; j < Numbers[2].size(); j++)
{
   if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
   {
      dots1[2].push_back(Numbers[2][j]);
   }
   else
   {
      dots1[1].push_back(Numbers[2][j]);
   }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
   if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
   {
      dots1[3].push_back(Numbers[3][j]);
   }
```

```
            else
            {
                dots1[2].push_back(Numbers[3][j]);
            }
        }
        Numbers.clear();

        dots1 = Sort_lenghts(dots1, lenghtS);
        for (int i = 0; i < dots1[1].size()-1; i++)
        {
            S += area_of_triangle_Ger(c, dots1[1][i+1], dots1[1][i]);
        }
        S += area_of_triangle_Ger(c, b, d);
        for (int i = 0; i < dots1[2].size()-1; i++)
        {
            S += area_of_triangle_Ger(d, dots1[2][i + 1], dots1[2][i]);
        }
    }
    else
    {
        dots1 = { {max_elm(Dot[i], 0)}, {max_elm(Dot[i], 1)} , {min_elm(Dot[i], 0)}, {min_elm(Dot[i ], 1)} };
        S += area_of_triangle_Ger(dots1[1][0], dots1[2][0], dots1[3][0]);
    }
    if (Dot[i + 1].size() > 4)
    {
        a = max_elm(Dot[i + 1], 0);
        b = min_elm(Dot[i + 1], 0);
```

```
        c = max_elm(Dot[i + 1], 1);
        d = min_elm(Dot[i + 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };

        Numbers = { {},{},{},{} };



        for (int j = 0; j < Dot[i + 1].size(); j++) //Переписать
        {
           if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
              Numbers[Sort_one(a, b, c, d, Dot[i + 1][j], lenghtS)].push_back(Dot[i
+ 1][j]);

        }

        for (int j = 0; j < Numbers[0].size(); j++)
        {
           if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
           {
              dots2[3].push_back(Numbers[0][j]);
           }

        }

        for (int j = 0; j < Numbers[1].size(); j++)
        {
           if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
           {
```

```
        dots2[1].push_back(Numbers[1][j]);
    }

}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
    {
        dots2[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots2[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
    else
    {
        dots2[2].push_back(Numbers[3][j]);
    }
}
```

```
            Numbers.clear();

            dots2 = Sort_lenghts(dots2, lenghtS);
            for (int i = 0; i < dots2[1].size() - 1; i++)
            {
               S -= area_of_triangle_Ger(c, dots2[1][i + 1], dots2[1][i]);
            }
            S -= area_of_triangle_Ger(c, b, d);
            for (int i = 0; i < dots2[2].size() - 1; i++)
            {
               S -= area_of_triangle_Ger(d, dots2[2][i + 1], dots2[2][i]);
            }

        }
        else
        {
            dots2 = { {max_elm(Dot[i+1], 1)}, {max_elm(Dot[i + 1], 0)} ,
{min_elm(Dot[i + 1], 1)}, {min_elm(Dot[i + 1], 0)} };
            S -= area_of_triangle_Ger(dots2[2][0], dots2[0][0], dots2[3][0]);
        }
        S += area_of_triangle_Ger(dots1[1][0], dots1[3][0], dots2[3][0]);
        S += area_of_triangle_Ger(dots2[1][0], dots2[3][0], dots1[1][0]);
    }

    a = max_elm(Dot[Dot.size() - 2], 0);
    b = min_elm(Dot[Dot.size() - 2], 0);
    c = max_elm(Dot[Dot.size() - 2], 1);
```

```
d = min_elm(Dot[Dot.size() - 2], 1);
dots1 = { {a},{c}, {b}, {d} };
if (Dot[Dot.size() - 2].size() > 4)
{
    Numbers = { {},{},{},{} };

    dots1 = { {a}, {c}, {b}, {d} };
    for (int j = 0; j < Dot[Dot.size() - 2].size(); j++) //Переписать
    {
        if (Dot[Dot.size() - 2][j] != a && Dot[Dot.size() - 2][j] != b &&
Dot[Dot.size() - 2][j] != c && Dot[Dot.size() - 2][j] != d)
            Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 2][j],
lenghtS)].push_back(Dot[Dot.size() - 2][j]);

    }

    for (int j = 0; j < Numbers[0].size(); j++)
    {
        if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
        {
            dots1[3].push_back(Numbers[0][j]);
        }

    }

    for (int j = 0; j < Numbers[1].size(); j++)
    {
        if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
        {
```

```
        dots1[1].push_back(Numbers[1][j]);
    }

}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
    {
        dots1[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots1[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots1[3].push_back(Numbers[3][j]);
    }
    else
    {
        dots1[2].push_back(Numbers[3][j]);
    }
}
```

```
        Numbers.clear();

        dots1 = Sort_lenghts(dots1, lenghtS);
        for (int i = 0; i < dots1[1].size() - 1; i++)
        {
            S += area_of_triangle_Ger(c, dots1[1][i + 1], dots1[1][i]);
        }
        S += area_of_triangle_Ger(c, b, d);
        for (int i = 0; i < dots1[2].size() - 1; i++)
        {
            S += area_of_triangle_Ger(d, dots1[2][i + 1], dots1[2][i]);
        }
    }
    else
    {
        dots1 = { {max_elm(Dot[Dot.size() - 2], 0)}, {max_elm(Dot[Dot.size() - 2],
1)} , {min_elm(Dot[Dot.size() - 2], 0)}, {min_elm(Dot[Dot.size() - 2], 1)} };
        S += area_of_triangle_Ger(dots1[1][0], dots1[2][0], dots1[3][0]);
    }

    if (Dot[Dot.size() - 1].size() > 4)
    {
        a = max_elm(Dot[Dot.size() - 1], 0);
        b = min_elm(Dot[Dot.size() - 1], 0);
        c = max_elm(Dot[Dot.size() - 1], 1);
        d = min_elm(Dot[Dot.size() - 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };
```

```
Numbers = { {},{},{},{} };



for (int j = 0; j < Dot[Dot.size() - 1].size(); j++) //Переписать
{
   if (Dot[Dot.size() - 1][j] != a && Dot[Dot.size() - 1][j] != b &&
Dot[Dot.size() - 1][j] != c && Dot[Dot.size() - 1][j] != d)
      Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 1][j],
lenghtS)].push_back(Dot[Dot.size() - 1][j]);


}


for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
   {
      dots2[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots2[3].push_back(Numbers[0][j]);
   }
}


for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
   {
      dots2[1].push_back(Numbers[1][j]);
```

```
    }
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) > lenghtS(Numbers[2][j], c))
    {
        dots2[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
}

Numbers.clear();

dots2 = Sort_lenghts(dots2, lenghtS);
for (int i = 0; i < dots2[3].size() - 1; i++)
{
```

```
            S += area_of_triangle_Ger(a, dots2[3][i+ 1], dots2[3][i]);
        }
        S += area_of_triangle_Ger(d, a, b);
        for (int i = 0; i < dots2[0].size() - 1; i++)
        {
            S +=  area_of_triangle_Ger(b, dots2[0][i + 1], dots2[0][i]);
        }
    }
    else
    {
        dots2 = { {max_elm(Dot[Dot.size() - 1], 1)}, {max_elm(Dot[Dot.size() - 1],
0)} , {min_elm(Dot[Dot.size() - 1], 1)}, {min_elm(Dot[Dot.size() - 1], 0)} };
        S +=   area_of_triangle_Ger(dots2[1][0], dots2[0][0], dots2[3][0]);
    }
    S += area_of_triangle_Ger(dots1[1][0], dots1[3][0], dots2[3][0]);
    S += area_of_triangle_Ger(dots2[1][0], dots2[3][0], dots1[1][0]);


    return S;
}


template<typename T>
T Vector_P_Nap(vector<T> A, vector<T> B)
{
    T Ans = 0;
    Ans += A[0]*B[1] - B[0] * A[1];
    return Ans;
}
```

```
template<typename T>
T Vector_P_D(vector<T> A, vector<T> B)
{
   T Ans = 0;
   Ans += abs(A[0] * B[1] - B[0] * A[1]);
   return Ans;
}

template<typename T>
T Vector_P_Nap_V(vector<T> A, vector<T> B)
{
   T Ans = 0;
   Ans += A[2] * B[1] - B[2] * A[1];
   return Ans;
}

template<typename T>
T Vector_P_D_V(vector<T> A, vector<T> B)
{
   T Ans = 0;
   Ans += abs(A[2] * B[1] - B[2] * A[1]);
   return Ans;
}

template<typename T>
T S_nap(vector < vector <vector<T>>>& dots)
{
   T S = 0;
```

```
    for (int i = 0; i < dots.size() - 1; i++)
    {
        for (int j = 0; j < dots[i].size() - 1; j++)
        {
            S += Vector_P_Nap(dots[i][j], dots[i][j + 1]);
        }
        S += Vector_P_Nap(dots[i][dots[i].size() - 1], dots[i +1 ][0]);
    }

    for (int j = 0; j < dots[dots.size() - 1].size() - 1; j++)
    {
        S += Vector_P_Nap(dots[dots.size() - 1][j], dots[dots.size() - 1][j + 1]);
    }
    S += Vector_P_Nap(dots[dots.size() - 1][dots[dots.size() - 1].size() - 1],
dots[0][0]);

    return 0.5*S;
}

template<typename T>
T S_nap2(vector < vector <vector<T>>>& dots, vector<T> V)
{
    T S = 0;

    vector <T> T1{ 0,0 };
    vector <T> T2{ 0,0 };
```

```
    for (int i = 0; i < dots.size() - 1; i++)
    {
        for (int j = 0; j < dots[i].size() - 1; j++)
        {
            T1[0] = dots[i][j][0] - V[0]; T1[1] = dots[i][j][1] - V[1];
            T2[0] = dots[i][j + 1][0] - V[0]; T2[1] = dots[i][j + 1][1] - V[1];
            S += Vector_P_D(T1, T2);
        }
        T1[0] = dots[i][dots[i].size() - 1][0] - V[0]; T1[1] = dots[i][dots[i].size() - 1][1] - V[1];
        T2[0] = dots[i + 1][0][0] - V[0]; T2[1] = dots[i + 1][0][1] - V[1];
        S += Vector_P_D(T1, T2);
    }

    for (int j = 0; j < dots[dots.size() - 1].size() - 1; j++)
    {
        T1[0] = dots[dots.size() - 1][j][0] - V[0]; T1[1] = dots[dots.size() - 1][j][1] - V[1];
        T2[0] = dots[dots.size() - 1][j + 1][0] - V[0]; T2[1] = dots[dots.size() - 1][j + 1][1] - V[1];
        S += Vector_P_D(T1, T2);
    }
    T1[0] = dots[dots.size() - 1][dots[dots.size() - 1].size() - 1][0] - V[0]; T1[1] = dots[dots.size() - 1][dots[dots.size() - 1].size() - 1][1] - V[1];
    T2[0] = dots[0][0][0] - V[0]; T2[1] = dots[0][0][1] - V[1];
    S += Vector_P_D(T1, T2);

    return 0.5 * S;
```

```
}

template<typename T>
T S_nap_V(vector < vector <vector<T>>>& dots)
{
   T S = 0;

   for (int i = 0; i < dots.size() - 1; i++)
   {
      for (int j = 0; j < dots[i].size() - 1; j++)
      {
         S += Vector_P_Nap_V(dots[i][j], dots[i][j + 1]);
      }
      S += Vector_P_Nap_V(dots[i][dots[i].size() - 1], dots[i + 1][0]);
   }

   for (int j = 0; j < dots[dots.size() - 1].size() - 1; j++)
   {
      S += Vector_P_Nap_V(dots[dots.size() - 1][j], dots[dots.size() - 1][j + 1]);
   }
   S += Vector_P_Nap_V(dots[dots.size() - 1][dots[dots.size() - 1].size() - 1],
dots[0][0]);

   return 0.5 * S;
}

template<typename T>
T S_nap2_V(vector < vector <vector<T>>>& dots, vector<T> V)
```

```
{
   T S = 0;

   vector <T> T1{ 0,0, 0 };
   vector <T> T2{ 0,0 ,0};

   for (int i = 0; i < dots.size() - 1; i++)
   {
      for (int j = 0; j < dots[i].size() - 1; j++)
      {
         T1[0] = dots[i][j][1] - V[1]; T1[1] = dots[i][j][2] - V[2];
         T2[0] = dots[i][j + 1][1] - V[1]; T2[1] = dots[i][j + 1][2] - V[2];
         S += Vector_P_D_V(T1, T2);
      }
      T1[1] = dots[i][dots[i].size() - 1][1] - V[1]; T1[2] = dots[i][dots[i].size() -
1][2] - V[2];
      T2[1] = dots[i + 1][0][1] - V[1]; T2[2] = dots[i + 1][0][2] - V[2];
      S += Vector_P_D_V(T1, T2);
   }

   for (int j = 0; j < dots[dots.size() - 1].size() - 1; j++)
   {
      T1[1] = dots[dots.size() - 1][j][1] - V[1]; T1[2] = dots[dots.size() - 1][j][2] -
V[2];
      T2[1] = dots[dots.size() - 1][j + 1][1] - V[1]; T2[2] = dots[dots.size() - 1][j +
1][2] - V[2];
      S += Vector_P_D_V(T1, T2);
   }
```

```
    T1[1] = dots[dots.size() - 1][dots[dots.size() - 1].size() - 1][1] - V[1]; T1[2] =
dots[dots.size() - 1][dots[dots.size() - 1].size() - 1][2] - V[2];
    T2[1] = dots[0][0][1] - V[1]; T2[2] = dots[0][0][2] - V[2];
    S += Vector_P_D_V(T1, T2);


    return 0.5 * S;
}


template<typename T>
T Cave_S2_2(vector < vector <vector<T>>> Dot)
{
    T S = 0;
    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector < vector < vector <T> > >  Numbers;
    vector<T> a, b, c, d;
    for (int i = 0; i < Dot.size() - 2; i++)
    {
        a = max_elm(Dot[i], 0);
        b = min_elm(Dot[i], 0);
        c = max_elm(Dot[i], 1);
        d = min_elm(Dot[i], 1);
        dots1 = { {a},{c}, {b}, {d} };
        if (Dot[i].size() > 4)
        {
            Numbers = { {},{},{},{} };


            dots1 = { {a}, {c}, {b}, {d} };
```

```
for (int j = 0; j < Dot[i].size(); j++) //Переписать
{
   if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c && Dot[i][j] != d)
      Numbers[Sort_one(a, b, c, d, Dot[i][j], lenghtS)].push_back(Dot[i][j]);

}

for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
   {
      dots1[3].push_back(Numbers[0][j]);
   }

}

for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
   {
      dots1[1].push_back(Numbers[1][j]);
   }

}

for (int j = 0; j < Numbers[2].size(); j++)
{
   if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
```

```
        {
            dots1[2].push_back(Numbers[2][j]);
        }
        else
        {
            dots1[1].push_back(Numbers[2][j]);
        }
    }

    for (int j = 0; j < Numbers[3].size(); j++)
    {
        if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
        {
            dots1[3].push_back(Numbers[3][j]);
        }
        else
        {
            dots1[2].push_back(Numbers[3][j]);
        }
    }
    Numbers.clear();

    dots1 = Sort_lenghts(dots1, lenghtS);
    dots1.erase(dots1.begin());
    dots1[2] = { d };
    S += S_nap(dots1);

}
```

```
else
{
   dots1 = { {max_elm(Dot[i], 0)}, {max_elm(Dot[i], 1)} , {min_elm(Dot[i], 0)}, {min_elm(Dot[i], 1)} };
   S += area_of_triangle_Ger(dots1[1][0], dots1[2][0], dots1[3][0]);
   dots1.erase(dots1.begin());
}
if (Dot[i + 1].size() > 4)
{
   a = max_elm(Dot[i + 1], 0);
   b = min_elm(Dot[i + 1], 0);
   c = max_elm(Dot[i + 1], 1);
   d = min_elm(Dot[i + 1], 1);
   dots2 = { {a}, {c}, {b}, {d} };

   Numbers = { {},{},{},{} };



   for (int j = 0; j < Dot[i + 1].size(); j++) //Переписать
   {
      if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c && Dot[i + 1][j] != d)
         Numbers[Sort_one(a, b, c, d, Dot[i + 1][j], lenghtS)].push_back(Dot[i + 1][j]);

   }

   for (int j = 0; j < Numbers[0].size(); j++)
   {
```

```
    if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
    {
        dots2[0].push_back(Numbers[0][j]);
    }
    else
    {
        dots2[3].push_back(Numbers[0][j]);
    }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
    if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
    {
        dots2[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) > lenghtS(Numbers[2][j], c))
    {
        dots2[1].push_back(Numbers[2][j]);
    }
```

```
        }

        for (int j = 0; j < Numbers[3].size(); j++)
        {
            if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
            {
                dots2[3].push_back(Numbers[3][j]);
            }
        }

        Numbers.clear();

        dots2 = Sort_lenghts(dots2, lenghtS);
        dots2.erase(dots2.begin());
        dots2[2] = { d };
        S -= S_nap(dots2);
    }
    else
    {
        dots2 = { {max_elm(Dot[i + 1], 1)}, {max_elm(Dot[i + 1], 0)} ,
{min_elm(Dot[i + 1], 1)}, {min_elm(Dot[i + 1], 0)} };
        S -= area_of_triangle_Ger(dots2[1][0], dots2[2][0], dots2[3][0]);
        dots2.erase(dots2.begin());
    }

    S += area_of_triangle_Ger(dots1[0][0], dots1[2][0], dots2[2][0]);
    S += area_of_triangle_Ger(dots2[0][0], dots2[2][0], dots1[0][0]);
```

```
}

a = max_elm(Dot[Dot.size() - 2], 0);
b = min_elm(Dot[Dot.size() - 2], 0);
c = max_elm(Dot[Dot.size() - 2], 1);
d = min_elm(Dot[Dot.size() - 2], 1);
dots1 = { {a},{c}, {b}, {d} };
if (Dot[Dot.size() - 2].size() > 4)
{
    Numbers = { {},{},{},{} };

    dots1 = { {a}, {c}, {b}, {d} };
    for (int j = 0; j < Dot[Dot.size() - 2].size(); j++) //Переписать
    {
        if (Dot[Dot.size() - 2][j] != a && Dot[Dot.size() - 2][j] != b &&
Dot[Dot.size() - 2][j] != c && Dot[Dot.size() - 2][j] != d)
            Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 2][j],
lenghtS)].push_back(Dot[Dot.size() - 2][j]);

    }

    for (int j = 0; j < Numbers[0].size(); j++)
    {
        if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
        {
            dots1[3].push_back(Numbers[0][j]);
        }
```

```
}

for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
   {
      dots1[1].push_back(Numbers[1][j]);
   }

}

for (int j = 0; j < Numbers[2].size(); j++)
{
   if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
   {
      dots1[2].push_back(Numbers[2][j]);
   }
   else
   {
      dots1[1].push_back(Numbers[2][j]);
   }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
   if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
   {
      dots1[3].push_back(Numbers[3][j]);
```

```
            }
            else
            {
                dots1[2].push_back(Numbers[3][j]);
            }
        }
        Numbers.clear();

        dots1 = Sort_lenghts(dots1, lenghtS);
        dots1.erase(dots1.begin());
        dots1[2] = { d };
        S += S_nap(dots1);

    }
    else
    {
        dots1 = { {max_elm(Dot[Dot.size() - 2], 0)}, {max_elm(Dot[Dot.size() - 2],
1)} , {min_elm(Dot[Dot.size() - 2], 0)}, {min_elm(Dot[Dot.size() - 2], 1)} };
        S += area_of_triangle_Ger(dots1[1][0], dots1[2][0], dots1[3][0]);
        dots1.erase(dots1.begin());
    }
    if (Dot[Dot.size() - 1].size() > 4)
    {
        a = max_elm(Dot[Dot.size() - 1], 0);
        b = min_elm(Dot[Dot.size() - 1], 0);
        c = max_elm(Dot[Dot.size() - 1], 1);
        d = min_elm(Dot[Dot.size() - 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };
```

```
Numbers = { {},{},{},{} };



for (int j = 0; j < Dot[Dot.size() - 1].size(); j++) //Переписать
{
   if (Dot[Dot.size() - 1][j] != a && Dot[Dot.size() - 1][j] != b &&
Dot[Dot.size() - 1][j] != c && Dot[Dot.size() - 1][j] != d)
      Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 1][j],
lenghtS)].push_back(Dot[Dot.size() - 1][j]);


}


for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
   {
      dots2[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots2[3].push_back(Numbers[0][j]);
   }
}


for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
   {
```

```
        dots2[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) > lenghtS(Numbers[2][j], c))
    {
        dots2[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
}

Numbers.clear();

dots2 = Sort_lenghts(dots2, lenghtS);
dots2.erase(dots2.begin());
```

```
        dots2[2] = { d };
        S += S_nap(dots2);
    }
    else
    {
        dots2 = { {max_elm(Dot[Dot.size() - 1], 1)}, {max_elm(Dot[Dot.size() - 1],
0)} , {min_elm(Dot[Dot.size() - 1], 1)}, {min_elm(Dot[Dot.size() - 1], 0)} };
        S += area_of_triangle_Ger(dots2[1][0], dots2[0][0], dots2[3][0]);
    }
    S += area_of_triangle_Ger(dots1[0][0], dots1[2][0], dots2[2][0]);
    S += area_of_triangle_Ger(dots2[0][0], dots2[2][0], dots1[0][0]);

    return S;
}

template<typename T>
T Cave_S2_3(vector < vector <vector<T>>> Dot)
{
    T S = 0;
    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector < vector < vector <T> > >  Numbers;
    vector<T> a, b, c, d;
    vector<T> V = { 0,0 };
    for (int i = 0; i < Dot.size() - 2; i++)
    {
        a = max_elm(Dot[i], 0);
        b = min_elm(Dot[i], 0);
```

```
c = max_elm(Dot[i], 1);
d = min_elm(Dot[i], 1);
dots1 = { {a},{c}, {b}, {d} };
if (Dot[i].size() > 4)
{
   Numbers = { {},{},{},{} };

   dots1 = { {a}, {c}, {b}, {d} };
   for (int j = 0; j < Dot[i].size(); j++) //Переписать
   {
      if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c && Dot[i][j] != d)
         Numbers[Sort_one(a, b, c, d, Dot[i][j], lenghtS)].push_back(Dot[i][j]);

   }

   for (int j = 0; j < Numbers[0].size(); j++)
   {
      if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
      {
         dots1[3].push_back(Numbers[0][j]);
      }

   }

   for (int j = 0; j < Numbers[1].size(); j++)
   {
      if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
      {
```

```
        dots1[1].push_back(Numbers[1][j]);
    }

}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
    {
        dots1[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots1[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots1[3].push_back(Numbers[3][j]);
    }
    else
    {
        dots1[2].push_back(Numbers[3][j]);
    }
}
```

```
        Numbers.clear();


        dots1 = Sort_lenghts(dots1, lenghtS);
        dots1.erase(dots1.begin());
        dots1[2] = { d };
        V[0] = (dots1[0][0][0] + dots1[1][0][0]) * 0.5; V[1] = (dots1[0][0][1] +
dots1[1][0][1]) * 0.5;
        S += S_nap2(dots1, V);


    }
    else
    {
        dots1 = { {max_elm(Dot[i], 0)}, {max_elm(Dot[i], 1)} , {min_elm(Dot[i],
0)}, {min_elm(Dot[i], 1)} };
        S += area_of_triangle_Ger(dots1[1][0], dots1[2][0], dots1[3][0]);
        dots1.erase(dots1.begin());
    }
    if (Dot[i + 1].size() > 4)
    {
        a = max_elm(Dot[i + 1], 0);
        b = min_elm(Dot[i + 1], 0);
        c = max_elm(Dot[i + 1], 1);
        d = min_elm(Dot[i + 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };


        Numbers = { {},{},{},{} };



        for (int j = 0; j < Dot[i + 1].size(); j++) //Переписать
```

```
        {
            if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
                Numbers[Sort_one(a, b, c, d, Dot[i + 1][j], lenghtS)].push_back(Dot[i
+ 1][j]);


        }


        for (int j = 0; j < Numbers[0].size(); j++)
        {
            if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
            {
                dots2[0].push_back(Numbers[0][j]);
            }
            else
            {
                dots2[3].push_back(Numbers[0][j]);
            }
        }


        for (int j = 0; j < Numbers[1].size(); j++)
        {
            if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
            {
                dots2[1].push_back(Numbers[1][j]);
            }
            else
            {
                dots2[0].push_back(Numbers[1][j]);
```

```
            }
        }

        for (int j = 0; j < Numbers[2].size(); j++)
        {
            if (lenghtS(Numbers[2][j], d) > lenghtS(Numbers[2][j], c))
            {
                dots2[1].push_back(Numbers[2][j]);
            }
        }

        for (int j = 0; j < Numbers[3].size(); j++)
        {
            if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
            {
                dots2[3].push_back(Numbers[3][j]);
            }
        }

        Numbers.clear();

        dots2 = Sort_lenghts(dots2, lenghtS);
        dots2.erase(dots2.begin());
        dots2[2] = { d };
        V[0] = (dots2[0][0][0] + dots2[1][0][0] ) * 0.5; V[1] = (dots2[0][0][1] +
dots2[1][0][1]) * 0.5;
        S -= S_nap2(dots2,V);
    }
```

```
        else
        {
            dots2 = { {max_elm(Dot[i + 1], 1)}, {max_elm(Dot[i + 1], 0)} ,
{min_elm(Dot[i + 1], 1)}, {min_elm(Dot[i + 1], 0)} };
            S -= area_of_triangle_Ger(dots2[1][0], dots2[2][0], dots2[3][0]);
            dots2.erase(dots2.begin());
        }

        S += area_of_triangle_Ger(dots1[0][0], dots1[2][0], dots2[2][0]);
        S += area_of_triangle_Ger(dots2[0][0], dots2[2][0], dots1[0][0]);



    }

    a = max_elm(Dot[Dot.size() - 2], 0);
    b = min_elm(Dot[Dot.size() - 2], 0);
    c = max_elm(Dot[Dot.size() - 2], 1);
    d = min_elm(Dot[Dot.size() - 2], 1);
    dots1 = { {a},{c}, {b}, {d} };
    if (Dot[Dot.size() - 2].size() > 4)
    {
        Numbers = { {},{},{},{} };

        dots1 = { {a}, {c}, {b}, {d} };
        for (int j = 0; j < Dot[Dot.size() - 2].size(); j++) //Переписать
        {
            if (Dot[Dot.size() - 2][j] != a && Dot[Dot.size() - 2][j] != b &&
Dot[Dot.size() - 2][j] != c && Dot[Dot.size() - 2][j] != d)
```

```
            Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 2][j],
lenghtS)].push_back(Dot[Dot.size() - 2][j]);


        }


        for (int j = 0; j < Numbers[0].size(); j++)
        {
            if (lenghtS(Numbers[0][j], c) > lenghtS(Numbers[0][j], d))
            {
                dots1[3].push_back(Numbers[0][j]);
            }


        }


        for (int j = 0; j < Numbers[1].size(); j++)
        {
            if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
            {
                dots1[1].push_back(Numbers[1][j]);
            }


        }


        for (int j = 0; j < Numbers[2].size(); j++)
        {
            if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
            {
                dots1[2].push_back(Numbers[2][j]);
```

```
        }
        else
        {
            dots1[1].push_back(Numbers[2][j]);
        }
    }

    for (int j = 0; j < Numbers[3].size(); j++)
    {
        if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
        {
            dots1[3].push_back(Numbers[3][j]);
        }
        else
        {
            dots1[2].push_back(Numbers[3][j]);
        }
    }
    Numbers.clear();

    dots1 = Sort_lenghts(dots1, lenghtS);
    dots1.erase(dots1.begin());
    dots1[2] = { d };
    V[0] = (dots1[0][0][0] + dots1[1][0][0] ) * 0.5; V[1] = (dots1[0][0][1] +
dots1[1][0][1] ) * 0.5;
    S += S_nap2(dots1,V);

}
```

```
    else
    {
        dots1 = { {max_elm(Dot[Dot.size() - 2], 0)}, {max_elm(Dot[Dot.size() - 2],
1)} , {min_elm(Dot[Dot.size() - 2], 0)}, {min_elm(Dot[Dot.size() - 2], 1)} };
        S += area_of_triangle_Ger(dots1[1][0], dots1[2][0], dots1[3][0]);
        dots1.erase(dots1.begin());
    }
    if (Dot[Dot.size() - 1].size() > 4)
    {
        a = max_elm(Dot[Dot.size() - 1], 0);
        b = min_elm(Dot[Dot.size() - 1], 0);
        c = max_elm(Dot[Dot.size() - 1], 1);
        d = min_elm(Dot[Dot.size() - 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };

        Numbers = { {},{},{},{} };



        for (int j = 0; j < Dot[Dot.size() - 1].size(); j++) //Переписать
        {
            if (Dot[Dot.size() - 1][j] != a && Dot[Dot.size() - 1][j] != b &&
Dot[Dot.size() - 1][j] != c && Dot[Dot.size() - 1][j] != d)
                Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 1][j],
lenghtS)].push_back(Dot[Dot.size() - 1][j]);


        }


        for (int j = 0; j < Numbers[0].size(); j++)
        {
```

```
    if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
    {
        dots2[0].push_back(Numbers[0][j]);
    }
    else
    {
        dots2[3].push_back(Numbers[0][j]);
    }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
    if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
    {
        dots2[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) > lenghtS(Numbers[2][j], c))
    {
        dots2[1].push_back(Numbers[2][j]);
    }
```

```
        }

        for (int j = 0; j < Numbers[3].size(); j++)
        {
            if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
            {
                dots2[3].push_back(Numbers[3][j]);
            }
        }

        Numbers.clear();

        dots2 = Sort_lenghts(dots2, lenghtS);
        dots2.erase(dots2.begin());
        dots2[2] = { d };
        V[0] = (dots2[0][0][0] + dots2[1][0][0] ) * 0.5; V[1] = (dots2[0][0][1] +
dots2[1][0][1]) * 0.5;
        S += S_nap2(dots2,V);
    }
    else
    {
        dots2 = { {max_elm(Dot[Dot.size() - 1], 1)}, {max_elm(Dot[Dot.size() - 1],
0)} , {min_elm(Dot[Dot.size() - 1], 1)}, {min_elm(Dot[Dot.size() - 1], 0)} };
        S += area_of_triangle_Ger(dots2[1][0], dots2[0][0], dots2[3][0]);
        dots2.erase(dots2.begin());
    }
    S += area_of_triangle_Ger(dots1[0][0], dots1[2][0], dots2[2][0]);
    S += area_of_triangle_Ger(dots2[0][0], dots2[2][0], dots1[0][0]);
```

```
    return S;
}



template<typename T>
T Cave_V2(vector < vector <vector<T>>> Dot)
{
    T V = 0;
    T S1 = 0;
    T S2 = 0;
    T H = 0;
    vector<T> a, b, c, d;

    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector < vector < vector <T> > > Numbers;
    vector<T> V1{ 0, 0, 0 }, V2{ 0, 0, 0 };


    for (int i = 0; i < Dot.size() - 2; i++)
    {

        if (Dot[i].size() > 4)
        {
            Numbers = { {},{},{},{} };
            a = max_elm(Dot[i], 2);
```

```
            b = min_elm(Dot[i], 2);
            c = max_elm(Dot[i], 1);
            d = min_elm(Dot[i], 1);
            dots1 = { {a}, {c}, {b}, {d} };
            dots1 = dots_original_V(dots1, Dot[i]);
            a = dots1[0][0]; c = dots1[1][0]; b = dots1[2][0]; d = dots1[3][0];
            for (int j = 0; j < Dot[i].size(); j++)
            {
               if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c && Dot[i][j] !=
d)
                  Numbers[Sort_one(a, b, c, d, Dot[i][j],
lenghtV)].push_back(Dot[i][j]);

            }

            for (int j = 0; j < Numbers[0].size(); j++)
            {
               if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
               {
                  dots1[0].push_back(Numbers[0][j]);
               }
               else
               {
                  dots1[3].push_back(Numbers[0][j]);
               }
            }

            for (int j = 0; j < Numbers[1].size(); j++)
            {
```

```
    if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
    {
        dots1[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots1[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots1[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots1[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
    {
        dots1[3].push_back(Numbers[3][j]);
    }
```

```
            else
            {
                dots1[2].push_back(Numbers[3][j]);
            }
        }

    }
    else
    {
        dots1 = { {max_elm(Dot[i], 2)}, {max_elm(Dot[i], 1)},
{min_elm(Dot[i], 2)}, {min_elm(Dot[i], 1)} };
        dots1 = dots_original_V(dots1, Dot[i]);
    }


    if (Dot[i + 1].size() > 4)
    {
        Numbers = { {},{},{},{} };
        a = max_elm(Dot[i + 1], 2);
        b = min_elm(Dot[i + 1], 2);
        c = max_elm(Dot[i + 1], 1);
        d = min_elm(Dot[i + 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };
        dots2 = dots_original_V(dots2, Dot[i + 1]);
        a = dots2[0][0]; c = dots2[1][0]; b = dots2[2][0]; d = dots2[3][0];
        for (int j = 0; j < Dot[i + 1].size(); j++)
        {
            if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
```

```
                Numbers[Sort_one(a, b, c, d, Dot[i + 1][j],
lenghtV)].push_back(Dot[i + 1][j]);


            }


            for (int j = 0; j < Numbers[0].size(); j++)
            {
               if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
               {
                  dots2[0].push_back(Numbers[0][j]);
               }
               else
               {
                  dots2[3].push_back(Numbers[0][j]);
               }
            }


            for (int j = 0; j < Numbers[1].size(); j++)
            {
               if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
               {
                  dots2[1].push_back(Numbers[1][j]);
               }
               else
               {
                  dots2[0].push_back(Numbers[1][j]);
               }
            }
```

```
    for (int j = 0; j < Numbers[2].size(); j++)
    {
        if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
        {
            dots2[2].push_back(Numbers[2][j]);
        }
        else
        {
            dots2[1].push_back(Numbers[2][j]);
        }
    }

    for (int j = 0; j < Numbers[3].size(); j++)
    {
        if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
        {
            dots2[3].push_back(Numbers[3][j]);
        }
        else
        {
            dots2[2].push_back(Numbers[3][j]);
        }
    }
    Numbers.clear();

}
else
```

```
        {
            dots2 = { {max_elm(Dot[i + 1], 2)}, {max_elm(Dot[i + 1], 1)} ,
{min_elm(Dot[i + 1], 2)}, {min_elm(Dot[i + 1], 1)} };
            dots2 = dots_original_V(dots2, Dot[i + 1]);
        }

        dots1 = Sort_lenghts(dots1, lenghtV);
        dots2 = Sort_lenghts(dots2, lenghtV);
        S1 = S_nap_V(dots1);
        S2 = S_nap_V(dots2);
        H = lenghtS(min_elm(Dot[i], 0), min_elm(Dot[i+1], 0));

        V +=  H*  (S1+ sqrt(S1*S2)+S2)/3;


        dots1.clear();
        dots2.clear();

    }

    if (Dot[Dot.size() - 2].size() > 4)
    {
        Numbers = { {},{},{},{} };
        a = max_elm(Dot[Dot.size() - 2], 2);
        b = min_elm(Dot[Dot.size() - 2], 2);
        c = max_elm(Dot[Dot.size() - 2], 1);
        d = min_elm(Dot[Dot.size() - 2], 1);
        dots1 = { {a}, {c}, {b}, {d} };
```

```
dots1 = dots_original_V(dots1, Dot[Dot.size() - 2]);
a = dots1[0][0]; c = dots1[1][0]; b = dots1[2][0]; d = dots1[3][0];
for (int j = 0; j < Dot[Dot.size() - 2].size(); j++)
{
   if (Dot[Dot.size() - 2][j] != a && Dot[Dot.size() - 2][j] != b &&
Dot[Dot.size() - 2][j] != c && Dot[Dot.size() - 2][j] != d)
      Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 2][j],
lenghtV)].push_back(Dot[Dot.size() - 2][j]);

}

for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
   {
      dots1[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots1[3].push_back(Numbers[0][j]);
   }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
   {
      dots1[1].push_back(Numbers[1][j]);
   }
```

```
    else
    {
        dots1[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots1[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots1[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
    {
        dots1[3].push_back(Numbers[3][j]);
    }
    else
    {
        dots1[2].push_back(Numbers[3][j]);
    }
```

```
        }


    }
    else
    {
        dots1 = { {max_elm(Dot[Dot.size() - 2], 2)}, {max_elm(Dot[Dot.size() - 2],
1)}, {min_elm(Dot[Dot.size() - 2], 2)}, {min_elm(Dot[Dot.size() - 2], 1)} };
        dots1 = dots_original_V(dots1, Dot[Dot.size() - 2]);
    }


    if (Dot[Dot.size() - 1].size() > 4)
    {
        Numbers = { {},{},{},{} };
        a = max_elm(Dot[Dot.size() - 1], 2);
        b = min_elm(Dot[Dot.size() - 1], 2);
        c = max_elm(Dot[Dot.size() - 1], 1);
        d = min_elm(Dot[Dot.size() - 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };
        dots2 = dots_original_V(dots2, Dot[Dot.size() - 1]);
        a = dots2[0][0]; c = dots2[1][0]; b = dots2[2][0]; d = dots2[3][0];
        for (int j = 0; j < Dot[Dot.size() - 1].size(); j++)
        {
            if (Dot[Dot.size() - 1][j] != a && Dot[Dot.size() - 1][j] != b &&
Dot[Dot.size() - 1][j] != c && Dot[Dot.size() - 1][j] != d)
                Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 1][j],
lenghtV)].push_back(Dot[Dot.size() - 1][j]);


        }
```

```
for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
   {
      dots2[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots2[3].push_back(Numbers[0][j]);
   }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
   {
      dots2[1].push_back(Numbers[1][j]);
   }
   else
   {
      dots2[0].push_back(Numbers[1][j]);
   }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
   if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
   {
```

```
            dots2[2].push_back(Numbers[2][j]);
        }
        else
        {
            dots2[1].push_back(Numbers[2][j]);
        }
    }

    for (int j = 0; j < Numbers[3].size(); j++)
    {
        if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
        {
            dots2[3].push_back(Numbers[3][j]);
        }
        else
        {
            dots2[2].push_back(Numbers[3][j]);
        }
    }
    Numbers.clear();

}
else
{
    dots2 = { {max_elm(Dot[Dot.size() - 1], 2)}, {max_elm(Dot[Dot.size() - 1],
1)} , {min_elm(Dot[Dot.size() - 1], 2)}, {min_elm(Dot[Dot.size() - 1], 1)} };
    dots2 = dots_original_V(dots2, Dot[Dot.size() - 1]);
}
```

```
    dots1 = Sort_lenghts(dots1, lenghtV);
    dots2 = Sort_lenghts(dots2, lenghtV);
    S1 = S_nap_V(dots1);
    S2 = S_nap_V(dots2);
    H = lenghtS(min_elm(Dot[Dot.size() - 2], 0), max_elm(Dot[Dot.size() - 1], 0));

    V += H * (S1 + sqrt(S1 * S2) + S2) / 3.0;
    return V;
}


template<typename T>
T Cave_V2_2(vector < vector <vector<T>>> Dot)
{
    T V = 0;
    T S1 = 0;
    T S2 = 0;
    T H = 0;
    vector<T> a, b, c, d;

    vector < vector < vector <T> > > dots1;
    vector < vector < vector <T> > > dots2;
    vector < vector < vector <T> > > Numbers;
    vector<T> V1{ 0,0, 0 }; vector<T> V2{ 0,0, 0 };


    for (int i = 0; i < Dot.size() - 2; i++)
    {
```

```
if (Dot[i].size() > 4)
{
   Numbers = { {},{},{},{} };
   a = max_elm(Dot[i], 2);
   b = min_elm(Dot[i], 2);
   c = max_elm(Dot[i], 1);
   d = min_elm(Dot[i], 1);
   dots1 = { {a}, {c}, {b}, {d} };
   dots1 = dots_original_V(dots1, Dot[i]);
   a = dots1[0][0]; c = dots1[1][0]; b = dots1[2][0]; d = dots1[3][0];
   for (int j = 0; j < Dot[i].size(); j++)
   {
      if (Dot[i][j] != a && Dot[i][j] != b && Dot[i][j] != c && Dot[i][j] != d)
         Numbers[Sort_one(a, b, c, d, Dot[i][j],
lenghtV)].push_back(Dot[i][j]);

   }

   for (int j = 0; j < Numbers[0].size(); j++)
   {
      if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
      {
         dots1[0].push_back(Numbers[0][j]);
      }
      else
      {
         dots1[3].push_back(Numbers[0][j]);
```

```
    }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
    if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
    {
        dots1[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots1[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots1[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots1[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
```

```
        {
            if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
            {
                dots1[3].push_back(Numbers[3][j]);
            }
            else
            {
                dots1[2].push_back(Numbers[3][j]);
            }
        }

    }
    else
    {
        dots1 = { {max_elm(Dot[i], 2)}, {max_elm(Dot[i], 1)}, {min_elm(Dot[i], 2)}, {min_elm(Dot[i], 1)} };
        dots1 = dots_original_V(dots1, Dot[i]);
    }


    if (Dot[i + 1].size() > 4)
    {
        Numbers = { {},{},{},{} };
        a = max_elm(Dot[i + 1], 2);
        b = min_elm(Dot[i + 1], 2);
        c = max_elm(Dot[i + 1], 1);
        d = min_elm(Dot[i + 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };
```

```
        dots2 = dots_original_V(dots2, Dot[i + 1]);
        a = dots2[0][0]; c = dots2[1][0]; b = dots2[2][0]; d = dots2[3][0];
        for (int j = 0; j < Dot[i + 1].size(); j++)
        {
           if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
               Numbers[Sort_one(a, b, c, d, Dot[i + 1][j], lenghtV)].push_back(Dot[i
+ 1][j]);


        }


        for (int j = 0; j < Numbers[0].size(); j++)
        {
           if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
           {
              dots2[0].push_back(Numbers[0][j]);
           }
           else
           {
              dots2[3].push_back(Numbers[0][j]);
           }
        }


        for (int j = 0; j < Numbers[1].size(); j++)
        {
           if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
           {
              dots2[1].push_back(Numbers[1][j]);
           }
```

```
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots2[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots2[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
    else
    {
        dots2[2].push_back(Numbers[3][j]);
    }
```

```
        }
        Numbers.clear();


    }
    else
    {
        dots2 = { {max_elm(Dot[i + 1], 2)}, {max_elm(Dot[i + 1], 1)} ,
{min_elm(Dot[i + 1], 2)}, {min_elm(Dot[i + 1], 1)} };
        dots2 = dots_original_V(dots2, Dot[i + 1]);
    }



    dots1 = Sort_lenghts(dots1, lenghtV);
    dots2 = Sort_lenghts(dots2, lenghtV);
    V1[2] = (dots1[1][0][2] + dots1[0][0][2] + dots1[2][0][2] + dots1[3][0][2]) *
0.25; V1[1] = (dots1[1][0][1] + dots1[0][0][1] + dots1[2][0][1] + dots1[3][0][1]) *
0.25;
    V2[2] = (dots1[1][0][2] + dots1[0][0][2] + dots1[2][0][2] + dots1[3][0][2]) *
0.25; V2[1] = (dots1[1][0][1] + dots1[0][0][1] + dots1[2][0][1] + dots1[3][0][1]) *
0.25;
    S1 = S_nap2_V(dots1, V1);
    S2 = S_nap2_V(dots2, V2);
    H = lenghtS(min_elm(Dot[i], 0), min_elm(Dot[i + 1], 0));


    V += H * (S1 + sqrt(S1 * S2) + S2) / 3;



    dots1.clear();
    dots2.clear();
```

```
}

if (Dot[Dot.size() - 2].size() > 4)
{
    Numbers = { {},{},{},{} };
    a = max_elm(Dot[Dot.size() - 2], 2);
    b = min_elm(Dot[Dot.size() - 2], 2);
    c = max_elm(Dot[Dot.size() - 2], 1);
    d = min_elm(Dot[Dot.size() - 2], 1);
    dots1 = { {a}, {c}, {b}, {d} };
    dots1 = dots_original_V(dots1, Dot[Dot.size() - 2]);
    a = dots1[0][0]; c = dots1[1][0]; b = dots1[2][0]; d = dots1[3][0];
    for (int j = 0; j < Dot[Dot.size() - 2].size(); j++)
    {
        if (Dot[Dot.size() - 2][j] != a && Dot[Dot.size() - 2][j] != b &&
Dot[Dot.size() - 2][j] != c && Dot[Dot.size() - 2][j] != d)
            Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 2][j],
lenghtV)].push_back(Dot[Dot.size() - 2][j]);


    }


    for (int j = 0; j < Numbers[0].size(); j++)
    {
        if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
        {
            dots1[0].push_back(Numbers[0][j]);
        }
        else
        {
```

```
        dots1[3].push_back(Numbers[0][j]);
    }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
    if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
    {
        dots1[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots1[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots1[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots1[1].push_back(Numbers[2][j]);
    }
}
```

```
        for (int j = 0; j < Numbers[3].size(); j++)
        {
            if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
            {
                dots1[3].push_back(Numbers[3][j]);
            }
            else
            {
                dots1[2].push_back(Numbers[3][j]);
            }
        }

    }
    else
    {
        dots1 = { {max_elm(Dot[Dot.size() - 2], 2)}, {max_elm(Dot[Dot.size() - 2],
1)}, {min_elm(Dot[Dot.size() - 2], 2)}, {min_elm(Dot[Dot.size() - 2], 1)} };
        dots1 = dots_original_V(dots1, Dot[Dot.size() - 2]);
    }

    if (Dot[Dot.size() - 1].size() > 4)
    {
        Numbers = { {},{},{},{} };
        a = max_elm(Dot[Dot.size() - 1], 2);
        b = min_elm(Dot[Dot.size() - 1], 2);
        c = max_elm(Dot[Dot.size() - 1], 1);
        d = min_elm(Dot[Dot.size() - 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };
```

```
dots2 = dots_original_V(dots2, Dot[Dot.size() - 1]);
a = dots2[0][0]; c = dots2[1][0]; b = dots2[2][0]; d = dots2[3][0];
for (int j = 0; j < Dot[Dot.size() - 1].size(); j++)
{
   if (Dot[Dot.size() - 1][j] != a && Dot[Dot.size() - 1][j] != b &&
Dot[Dot.size() - 1][j] != c && Dot[Dot.size() - 1][j] != d)
      Numbers[Sort_one(a, b, c, d, Dot[Dot.size() - 1][j],
lenghtV)].push_back(Dot[Dot.size() - 1][j]);

}

for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtV(Numbers[0][j], c) < lenghtV(Numbers[0][j], d))
   {
      dots2[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots2[3].push_back(Numbers[0][j]);
   }
}

for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtV(Numbers[1][j], b) < lenghtV(Numbers[1][j], a))
   {
      dots2[1].push_back(Numbers[1][j]);
   }
```

```
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtV(Numbers[2][j], d) < lenghtV(Numbers[2][j], c))
    {
        dots2[2].push_back(Numbers[2][j]);
    }
    else
    {
        dots2[1].push_back(Numbers[2][j]);
    }
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtV(Numbers[3][j], a) < lenghtV(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
    else
    {
        dots2[2].push_back(Numbers[3][j]);
    }
```

```
        }
        Numbers.clear();


    }
    else
    {
        dots2 = { {max_elm(Dot[Dot.size() - 1], 2)}, {max_elm(Dot[Dot.size() - 1],
1)} , {min_elm(Dot[Dot.size() - 1], 2)}, {min_elm(Dot[Dot.size() - 1], 1)} };
        dots2 = dots_original_V(dots2, Dot[Dot.size() - 1]);
    }
    dots1 = Sort_lenghts(dots1, lenghtV);
    dots2 = Sort_lenghts(dots2, lenghtV);
    V1[2] = (dots1[1][0][2] + dots1[0][0][2] + dots1[2][0][2] + dots1[3][0][2]) *
0.25; V1[1] = (dots1[1][0][1] + dots1[0][0][1] + dots1[2][0][1] + dots1[3][0][1]) *
0.25;
    V2[2] = (dots1[1][0][2] + dots1[0][0][2] + dots1[2][0][2] + dots1[3][0][2]) *
0.25; V2[1] = (dots1[1][0][1] + dots1[0][0][1] + dots1[2][0][1] + dots1[3][0][1]) *
0.25;
    S1 = S_nap2_V(dots1, V1);
    S2 = S_nap2_V(dots2, V2);
    H = lenghtS(min_elm(Dot[Dot.size() - 2], 0), min_elm(Dot[Dot.size() - 1], 0));


    V += H * (S1 + sqrt(S1 * S2) + S2) / 3;


    return V;
}


template<typename T>
T Cave_V3elN1(vector < vector <vector<T>>> Dot)
{
```

```
T S = 0;
T C = 0;
T V = 0;
T coef = 2.0 / 3;
vector < vector < vector <T> > >  dots2;
vector < vector < vector <T> > >  Numbers;
vector< vector <vector<T>>> Temp;
vector<T> a, b, c, d;
for (int i = 0; i < Dot.size() - 2; i++)
{
    Temp = { Dot[i], Dot[i + 1] };
    S = Cave_S2_2(Temp);
    C = (sqrt((max_elm(Dot[i], 2)[2] - min_elm(Dot[i], 2)[2]) * (max_elm(Dot[i], 2)[2] - min_elm(Dot[i], 2)[2])) +
        sqrt((max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]) * (max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]))) *
        0.5;
    V += coef * S * C;

    if (Dot[i + 1].size() > 4)
    {
        a = max_elm(Dot[i + 1], 0);
        b = min_elm(Dot[i + 1], 0);
        c = max_elm(Dot[i + 1], 1);
        d = min_elm(Dot[i + 1], 1);
        dots2 = { {a}, {c}, {b}, {d} };

        Numbers = { {},{},{},{} };
```

```
        for (int j = 0; j < Dot[i + 1].size(); j++)
        {
            if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
                Numbers[Sort_one(a, b, c, d, Dot[i + 1][j], lenghtS)].push_back(Dot[i
+ 1][j]);


        }


        for (int j = 0; j < Numbers[0].size(); j++)
        {
            if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
            {
                dots2[0].push_back(Numbers[0][j]);
            }
            else
            {
                dots2[3].push_back(Numbers[0][j]);
            }
        }


        for (int j = 0; j < Numbers[1].size(); j++)
        {
            if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
            {
                dots2[1].push_back(Numbers[1][j]);
            }
            else
```

```
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
    {
        dots2[2].push_back(Numbers[2][j]);
    }
    else
        dots2[1].push_back(Numbers[2][j]);
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
    else
        dots2[2].push_back(Numbers[3][j]);



}

Numbers.clear();
```

```
            dots2 = Sort_lenghts(dots2, lenghtS);
            S = S_nap(dots2);
        }
        else
        {
            dots2 = { {max_elm(Dot[i + 1], 1)}, {max_elm(Dot[i + 1], 0)} ,
{min_elm(Dot[i + 1], 1)}, {min_elm(Dot[i + 1], 0)} };
            S = area_of_triangle_Ger(dots2[1][0], dots2[2][0], dots2[3][0]) +
area_of_triangle_Ger(dots2[1][0], dots2[0][0], dots2[3][0]);


        }
        C = (sqrt((max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]) *
(max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]))) *
            0.5;
        V -= coef * S * C;
    }
    Temp = { Dot[Dot.size() - 2],Dot[Dot.size() - 1] };
    S = Cave_S2_2(Temp);
    C = (sqrt((max_elm(Dot[Dot.size() - 2], 2)[2] - min_elm(Dot[Dot.size() - 2],
2)[2]) * (max_elm(Dot[Dot.size() - 2], 2)[2] - min_elm(Dot[Dot.size() - 2], 2)[2]))
+
        sqrt((max_elm(Dot[Dot.size() - 1], 2)[2] - min_elm(Dot[Dot.size() - 1], 2)[2])
* (max_elm(Dot[Dot.size() - 1], 2)[2] - min_elm(Dot[Dot.size() - 1], 2)[2]))) *
        0.5;
    V += coef * S * C;
    return V;
}
```

```
template<typename T>
T Cave_V3elN2(vector < vector <vector<T>>> Dot)
{
   T S = 0;
   T C = 0;
   T V = 0;
   T coef = 2.0 / 3;
   vector < vector < vector <T> > >  dots2;
   vector < vector < vector <T> > >  Numbers;
   vector< vector <vector<double>>> Temp;
   vector<T> a, b, c, d;
   for (int i = 0; i < Dot.size() - 2; i++)
   {
      Temp = { Dot[i], Dot[i + 1] };
      S = Cave_S2(Temp);
      C = (sqrt((max_elm(Dot[i], 2)[2] - min_elm(Dot[i], 2)[2]) * (max_elm(Dot[i],
2)[2] - min_elm(Dot[i], 2)[2])) +
         sqrt((max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]) *
(max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]))) *
         0.5;
      V += coef * S * C;

      if (Dot[i + 1].size() > 4)
      {
         a = max_elm(Dot[i + 1], 0);
         b = min_elm(Dot[i + 1], 0);
         c = max_elm(Dot[i + 1], 1);
         d = min_elm(Dot[i + 1], 1);
         dots2 = { {a}, {c}, {b}, {d} };
```

```
Numbers = { {},{},{},{} };



for (int j = 0; j < Dot[i + 1].size(); j++)
{
   if (Dot[i + 1][j] != a && Dot[i + 1][j] != b && Dot[i + 1][j] != c &&
Dot[i + 1][j] != d)
      Numbers[Sort_one(a, b, c, d, Dot[i + 1][j], lenghtS)].push_back(Dot[i
+ 1][j]);


}


for (int j = 0; j < Numbers[0].size(); j++)
{
   if (lenghtS(Numbers[0][j], c) < lenghtS(Numbers[0][j], d))
   {
      dots2[0].push_back(Numbers[0][j]);
   }
   else
   {
      dots2[3].push_back(Numbers[0][j]);
   }
}


for (int j = 0; j < Numbers[1].size(); j++)
{
   if (lenghtS(Numbers[1][j], b) < lenghtS(Numbers[1][j], a))
   {
```

```
        dots2[1].push_back(Numbers[1][j]);
    }
    else
    {
        dots2[0].push_back(Numbers[1][j]);
    }
}

for (int j = 0; j < Numbers[2].size(); j++)
{
    if (lenghtS(Numbers[2][j], d) < lenghtS(Numbers[2][j], c))
    {
        dots2[2].push_back(Numbers[2][j]);
    }
    else
        dots2[1].push_back(Numbers[2][j]);
}

for (int j = 0; j < Numbers[3].size(); j++)
{
    if (lenghtS(Numbers[3][j], a) < lenghtS(Numbers[3][j], b))
    {
        dots2[3].push_back(Numbers[3][j]);
    }
    else
        dots2[2].push_back(Numbers[3][j]);
```

```
        }

        Numbers.clear();

        dots2 = Sort_lenghts(dots2, lenghtS);
        for (int i = 0; i < dots2[1].size() - 1; i++)
        {
           S += area_of_triangle_Ger(c, dots2[1][i + 1], dots2[1][i]);
        }
        S += area_of_triangle_Ger(c, b, d);
        for (int i = 0; i < dots2[2].size() - 1; i++)
        {
           S += area_of_triangle_Ger(d, dots2[2][i + 1], dots2[2][i]);
        }
      }
      else
      {
        dots2 = { {max_elm(Dot[i + 1], 1)}, {max_elm(Dot[i + 1], 0)} ,
{min_elm(Dot[i + 1], 1)}, {min_elm(Dot[i + 1], 0)} };
        S = area_of_triangle_Ger(dots2[1][0], dots2[2][0], dots2[3][0]) +
area_of_triangle_Ger(dots2[1][0], dots2[0][0], dots2[3][0]);

      }
      C = (sqrt((max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2]) *
(max_elm(Dot[i + 1], 2)[2] - min_elm(Dot[i + 1], 2)[2])) *
        0.5;
      V -= coef * S * C;
   }
   Temp = { Dot[Dot.size() - 2],Dot[Dot.size() - 1] };
```

```
    S = Cave_S2(Temp);
    C = (sqrt((max_elm(Dot[Dot.size() - 2], 2)[2] - min_elm(Dot[Dot.size() - 2],
2)[2]) * (max_elm(Dot[Dot.size() - 2], 2)[2] - min_elm(Dot[Dot.size() - 2], 2)[2]))
+
        sqrt((max_elm(Dot[Dot.size() - 1], 2)[2] - min_elm(Dot[Dot.size() - 1], 2)[2])
* (max_elm(Dot[Dot.size() - 1], 2)[2] - min_elm(Dot[Dot.size() - 1], 2)[2]))) *
        0.5;
    V += coef * S * C;
    return V;
}
```