Технологии виртуальной реальности

2022/23 учебный год

Второй отборочный этап

Задача IV.1. Треугольник на цилиндре (100 баллов)

Темы: геометрия.

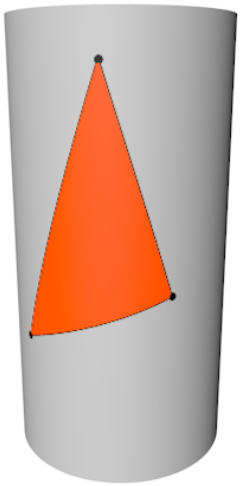
Условие

Заказчик выпускает 3-х мерный цилиндр $x^2 + y^2 = 1$ на котором отмечены три точки (x_i, y_i, z_i), $i = 1, 2, 3$.

Юному программисту Васе поручено вырезать треугольник из цветной бумаги так, чтобы при наклеивании этого треугольника на цилиндр его вершины совпали с отмеченными точками, и чтобы такой треугольник не был наклеен сам на себя (при этом допускается пересечение по точкам или наличие общих сторон).

В дальнейшем, из этого цилиндра сделают развёртку путём разрезания цилиндра по любой прямой параллельной оси Z и разворачивания его на плоскость. Заказчик требует, чтобы любая такая прямая сечения пересекала этот треугольник не более 2-х раз. Одним пересечением будем считать «заход» прямой в треугольник и «выход» из него.

Вася хочет минимизировать расход бумаги и поэтому собирается написать программу, которая по заданным 3 точкам на цилиндре определит наименьшую возможную площадь треугольника S, который удовлетворяет всем указанным требованиям. Помогите Васе написать такую программу.



Формат входных данных

В i-ой $(i = 1, 2, 3)$ строке содержатся вещественные числа x_i, y_i, z_i — координаты точек на цилиндре.

Формат выходных данных

Выходной файл содержит одно число S — минимальную возможную площадь треугольника с точностью до 5 знака после запятой. Если такой треугольник построить нельзя — выведите 0.

Ограничения

Для любых $x_i, y_i, z_i, i = 1, 2, 3$ выполнено $x_i^2 + y_i^2 = 1$.

Примеры

Пример №1

Стандартный ввод
1 0 1 1 0 0 0 1 0
Стандартный вывод
0.78540

Пример №2

Стандартный ввод
0.70711 0.70711 4 0.70711 0.70711 0 0.70711 -0.70711 0
Стандартный вывод
3.14159

Ссылка на архив с проверочными тестами: https://disk.yandex.ru/d/H7azTsnJRWIEWA.

Пример программы-решения

Ниже представлено решение на языке C++.

```
#include <iomanip>
#include <fstream>
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;

struct v3 {
  double x, y, z;
};
struct v2 {
  double phi, z;
};

int sign(double a) {
```

```
16    if (a > 0) {
17      return 1;
18    }
19    else if (a == 0) {
20      return 0;
21    }
22    else {
23      return -1;
24    }
25  }
26
27  int inTriangle(v2 p[3], v2 d) {
28    int s[3];
29    s[0] = sign((p[0].phi - d.phi) * (p[1].z - p[0].z) - (p[1].phi - p[0].phi) *
      ↪  (p[0].z - d.z));
30    s[1] = sign((p[1].phi - d.phi) * (p[2].z - p[1].z) - (p[2].phi - p[1].phi) *
      ↪  (p[1].z - d.z));
31    s[2] = sign((p[2].phi - d.phi) * (p[0].z - p[2].z) - (p[0].phi - p[2].phi) *
      ↪  (p[2].z - d.z));
32    if ((s[0] == -1 && s[1] == -1 && s[2] == -1) || (s[0] == 1 && s[1] == 1 && s[2]
      ↪  == 1)) {
33      return 1;
34    }
35    else if (s[0] == 0 || s[1] == 0 || s[2] == 0) {
36      bool flag = true;
37      for (int i = 1; i < 4; i++) {
38        if (s[i] == 0) {
39          if (((p[i % 3].phi <= d.phi && d.phi <= p[(i + 1) % 3].phi) || (p[(i + 1)
            ↪  % 3].phi <= d.phi && d.phi <= p[i % 3].phi)) &&
40            ((p[i % 3].z <= d.z && d.z <= p[(i + 1) % 3].z) || (p[(i + 1) % 3].z <=
              ↪  d.z && d.z <= p[i % 3].z))) {
41            if (!(p[i % 3].phi == d.phi && p[i % 3].z == d.z || p[(i + 1) % 3].phi
              ↪  == d.phi && p[(i + 1) % 3].z == d.z)) {
42              return 0;
43            }
44          }
45        }
46      }
47      return -1;
48    }
49    else {
50      return -1;
51    }
52  }
53
54  int main() {
55    ifstream fin("input.txt");
56    ofstream fout("output.txt");
57
58    v3 v;
59    v2 p[2][3];
60
61    double minS = 0;
62    for (int i = 0; i < 3; i++) {
63      fin >> v.x >> v.y >> v.z;
64      p[0][i].phi = atan2(v.y, v.x);
65      if (p[0][i].phi < 0) {
66        p[0][i].phi += M_PI * 2;
67      }
68      p[0][i].z = v.z;
```

```
69      p[1][i].phi = M_PI * 2 + p[0][i].phi;
70      p[1][i].z = p[0][i].z;
71    }
72    for (int a = 0; a < 2; a++) {
73      for (int b = 0; b < 2; b++) {
74        for (int c = 0; c < 2; c++) {
75          v2 cp[3];
76          cp[0] = p[a][0];
77          cp[1] = p[b][1];
78          cp[2] = p[c][2];
79          if (inTriangle(cp, p[(a + 1) % 2][0]) == 1 || inTriangle(cp, p[(b + 1) %
            ↪  2][1]) == 1 || inTriangle(cp, p[(c + 1) % 2][2]) == 1) {
80            continue;
81          }
82          if ((inTriangle(cp, p[(a + 1) % 2][0]) == 0 && inTriangle(cp, p[(b + 1) %
            ↪  2][1]) == 0) ||
83            (inTriangle(cp, p[(b + 1) % 2][1]) == 0 && inTriangle(cp, p[(c + 1) %
              ↪  2][2]) == 0) ||
84            (inTriangle(cp, p[(a + 1) % 2][0]) == 0 && inTriangle(cp, p[(c + 1) %
              ↪  2][2]) == 0)) {
85            continue;
86          }
87          double lenght[3];
88          double halfper = 0;
89          for (int i = 0; i < 3; i++) {
90            lenght[i] = sqrt(pow(cp[(i + 1) % 3].phi - cp[i].phi, 2) + pow(cp[(i +
              ↪  1) % 3].z - cp[i].z, 2));
91            halfper += lenght[i];
92          }
93          bool stop = false;
94          for (int i = 0; i < 3; i++) {
95            if (lenght[(i + 1) % 3] >= lenght[(i + 2) % 3] + lenght[(i + 3) % 3]) {
96              stop = true;
97              break;
98            }
99          }
100         if (stop) {
101           continue;
102         }
103         halfper /= 2;
104         double S = sqrt(halfper*(halfper - lenght[0])*(halfper -
            ↪  lenght[1])*(halfper - lenght[2]));
105         if ((S < minS || minS == 0) && abs(S) > 1.e-3) {
106           minS = S;
107         }
108       }
109     }
110   }
111   fout << fixed << setprecision(5) << minS;
112 }
```

Задача IV.2. Битый шлем (100 баллов)

Темы: 3D-моделирование.

Условие

Вам дана модель VR-шлема с некорректной топологией, которую вам надо исправить. А именно, надо избавить от двойных вершин и граней с вывернутыми нормалями. Две вершины считаются двойными, если находятся друг от друга на расстоянии не более 0,0001.

Имеются рендеры из `viewport` с нескольких ракурсов. Обратите внимание, что рендеры также несут дополнительную информацию о форме, расположении и размерах модели. Модель не должна содержать текстур.

Модель проверяется с помощью `add-on CheckToolBox` и попиксельного сравнения рендеров с указанных во входном файле ресурсов. К модели применяется автоматически созданный материал со случайно подобранным оттенком. В качестве метрики для сравнения рендеров моделей используется величина `dssim` по каждому цветовому каналу. Баллы за каждый тест начисляются в зависимости от величины метрики и полученного из аддона результата.

Формат входных данных

Во входном файле содержатся x, y, z — координаты камеры в метрах и rx, ry, rz — углы поворота в радианах.

Формат выходных данных

В качестве решения следует отправлять файл формата OBJ(расширение `.obj`). Размер файла не должен превышать 999997 Байт. Единицы измерения должны соответствовать физическим величинам. Координаты вершин модели должны быть указаны в метрах.

Примеры

Пример №1

Стандартный ввод
`7.35889 -6.92579 4.95831` `1.10932 0.0 0.814928`
Стандартный вывод
`см. рис. 1`

Пример №2

Стандартный ввод
`8.0155 8.89959 6.23837` `1.07406 0.0 2.38341`
Стандартный вывод
`см. рис. 2`

Решение

Решением является файл 3D модели в формате `obj`, доступен по ссылке: https://disk.yandex.ru/d/c42sjX2hHtgGdQ.

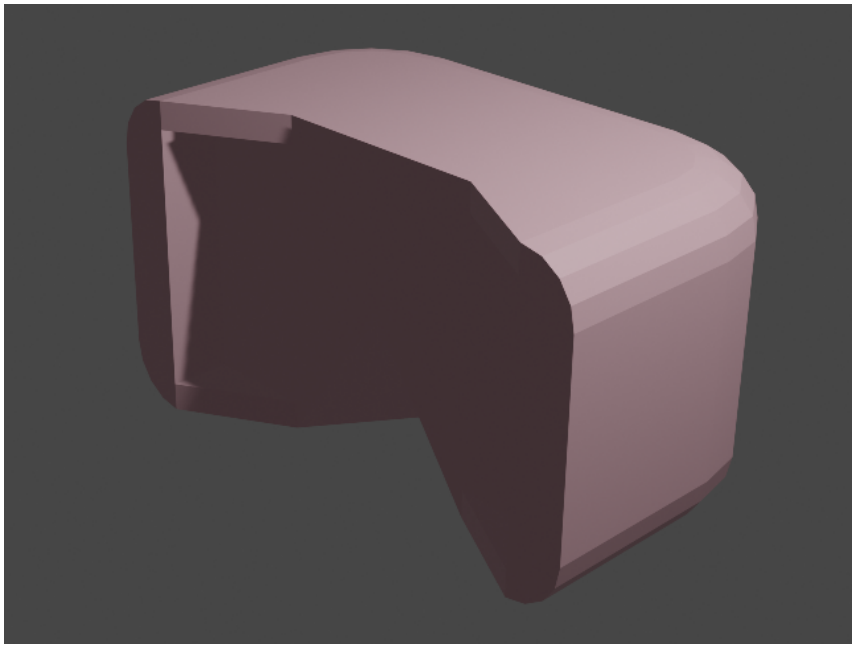


Рис. 1

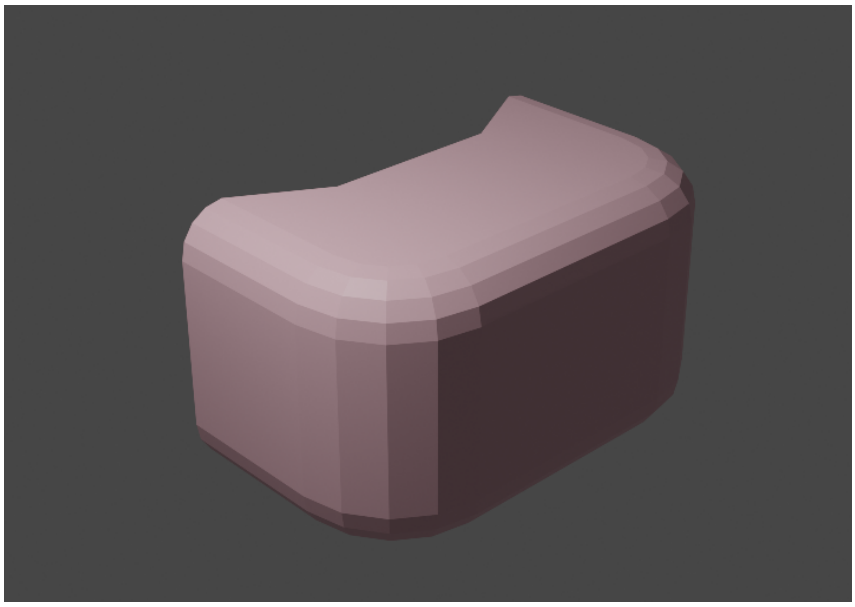


Рис. 2

Задача IV.3. Разрезание многогранника (100 баллов)

Темы: геометрия.

Условие

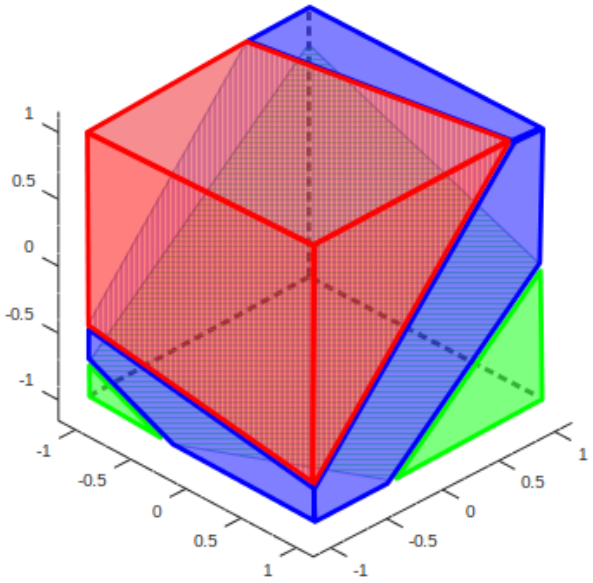
Пусть имеется выпуклый многогранник и две плоскости, каждая из которых задается ортогональным к ней вектором $\overrightarrow{U}_k$.

Известно, что плоскость можно сдвигать относительно начала координат путем смещения всех составляющих ее точек на вектор:

$$\overrightarrow{S}_k = t_k \cdot \frac{\overrightarrow{U}_k}{|\overrightarrow{U}_k|},$$

где t_k — произвольный скалярный параметр.

Требуется расположить указанные плоскости таким образом, чтобы они разбивали исходный многогранник ровно на три части одинакового объема.



Формат входных данных

В начале входного файла «`input.txt`» записано число N, за которым следует $3 \times N$ вещественных координат вершин.

Далее указано число M, за которым следует M граней, записанных в следующем виде.

Вначале для каждой грани указано число ее вершин, за которым следуют их индексы, перечисленные в порядке их обхода для получения ограничивающего цикла грани. При этом полагается, что нумерация вершин начинается с нуля.

В окончании входного файла записаны компоненты векторов, задающих плоскости (сначала для 1-го, а затем для 2-го).

Формат выходных данных

Если решение существует, в выходной файл «`output.txt`» следует вывести число 1 и пару вещественных параметров t_1 и t_2, указанных с точностью до 5-го знака после запятой.

В противном случае, выходной файл должен содержать единственное число 0.

Ограничения

Исходный многогранник является невырожденным (имеет ненулевой объем). Все грани представляют собой выпуклые многоугольники. Координаты всех вершин по модулю не превосходят 10. Число вершин и граней не превосходит 100. Гарантируется, что в пределах заданной точности ответ может быть получен однозначно.

Примеры

Пример №1

Стандартный ввод

```
8
 1.00000 -1.00000 -1.00000
 1.00000  1.00000 -1.00000
 1.00000  1.00000  1.00000
 1.00000 -1.00000  1.00000
-1.00000 -1.00000 -1.00000
-1.00000  1.00000 -1.00000
-1.00000  1.00000  1.00000
-1.00000 -1.00000  1.00000

6
4 0 1 2 3
4 7 6 5 4
4 5 6 2 1
4 0 3 7 4
4 7 3 2 6
4 5 1 0 4

-0.25000  1.50000 -1.50000
-0.75000  1.50000 -2.00000
```

Стандартный вывод

```
1
-0.25972  0.26752
```

Решение

Обрезка выпуклого многогранника

Для начала разберемся с алгоритмом, выполняющим обрезку выпуклого многогранника с выпуклыми гранями заданной плоскостью.

Разделим вершины исходного многогранника относительно секущей плоскости:

1. внутренние — вершины, лежащие в заданной плоскости (с учетом погрешности округления);
2. верхние — вершины, лежащие выше заданной плоскости;
3. нижние — вершины, лежащие ниже заданной плоскости.

Для этого достаточно будет проверить знак скалярного произведения

$$(X_i - C_i) \cdot U,$$

где X_i — координаты текущей вершины;

C — произвольная точка плоскости (с учетом ее смещения);

U — нормаль к плоскости.

Все внутренние и верхние вершины добавляем в результирующий список P.

Далее из ребер многогранника выделим следующие разновидности:

1. внутренние — у которых обе вершины являются внутренними;
2. секущие — если одна вершина верхняя, а другая нижняя;
3. верхние — если есть верхняя, но нет нижней вершины;
4. нижние — если есть нижняя, но нет верхней вершины.

На каждом секущем ребре определим точку, в которой оно пересекает плоскость, и сформируем на ее основе новую вершину.

Само ребро при этом укорачиваем путем замены его нижней вершины на новую.

Отдельно для каждого ребра запоминаем список его вершин, лежащих в плоскости (включая новую).

Добавляем новую вершину в результирующий список вершин P. Все ребра, кроме нижних, добавляем в новый список E.

Далее рассмотрим двумерные грани и разделим их на:

1. внутренние — у которых все ребра являются внутренними;
2. секущие — есть секущее ребро, либо пара верхнее-нижнее;
3. верхние — есть верхнее, но нет секущих/нижних ребер;
4. нижние — есть нижнее, но нет секущих/верхних ребер.

Обратим внимание, что т. к. все грани выпуклые, то они не могут иметь секущее и внутреннее ребро одновременно.

Наличие внутреннего ребра также исключает существование пары из верхнего и нижнего ребер.

При этом секущая плоскость может проходить через последовательность внутренних ребер.

Можно также утверждать, что секущих ребер не может быть больше двух.

Руководствуясь этой логикой, пройдемся по ребрам секущей грани, собирая ранее приписанные к ним вершины.

Таких вершин всегда будет две. Сформируем из них новое ребро.

Саму грань при этом обрезаем, удалив из нее нижние ребра. Вместо них вставляем новое.

Для каждой грани запоминаем список ее ребер, лежащих в плоскости (включая новое).

Добавим новое ребро в результирующий список ребер E. Все грани, кроме нижних, добавляем в список F.

В завершение, если имелась секущая грань, либо пара верхняя-нижняя, собираем лежащие в плоскости ребра и формируем новую грань.

В силу выпуклости, такая грань всегда будет одна.

Объем выпуклого многогранника

Чтобы посчитать объем полученного тела, достаточно будет разложить его на составляющие тетраэдры.

Для этого вычислим геометрический центр вершин многогранника, а также центры каждой из граней.

Далее, запустим обход ребер на каждой из граней. Порядок обхода ребер здесь роли не играет. Дополняя такое ребро центральной точкой грани и центром многогранника, получим тетраэдр, объем которого можно найти по известной формуле.

Разбиение многогранника

Теперь рассмотрим способ разбиения многогранника объемом V_0 на три равные части.

Выберем 1-ю плоскость, и разобьем наше тело на две части объемом $1/3$ и $2/3$ от V_0 соответственно.

Будем искать соответствующий параметр t, отвечающий за смещение плоскости вдоль ее нормали.

Определим для него диапазон, на котором смещенная плоскость будет пересекать тело, найдя минимум и максимум из проекций вершин на соответствующий вектор нормали.

В указанном диапазоне объем отсеченного тела будет монотонно расти от нуля до V_0, что дает нам возможность воспользоваться методом двоичного поиска. После того, как нужное значение t было найдено, получим два тела: меньшее B_1 и большее B_2.

Задействуем вторую плоскость, и найдем интервал, на котором она пересекает B_2. Исключим из него подынтервал, на котором она пересекает тело B_1. Если этого не сделать, исходное тело может оказаться разбитым более чем на три части. Запускаем аналогичный поиск, чтобы разбить тело B_2 на две равные по объему части. При этом, в отличие от первого разбиения, здесь поиск может завершиться неудачей, если желаемый ответ окажется за пределами допустимого интервала.

Обратим внимание на то, что направление нормали 1-й плоскости определяет часть тела, которую мы будем пытаться разбить 2-й плоскостью. Поэтому в случае неудачи нормаль можно будет инвертировать и попробовать еще раз.

Пример программы-решения

Ниже представлено решение на языке C++.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <limits>
#include <exception>
#include <algorithm>
#include <vector>
#include <array>
#include <cmath>
#include <map>
#include <set>
```

```
12
13  template <typename T, unsigned N> using t_tuple = std::array<T, N>;
14  template <typename T> using t_list = std::vector<T>;
15
16  typedef double t_scalar; typedef int t_index;
17
18  typedef t_tuple<t_scalar, 3> t_vector;
19  typedef t_tuple<t_scalar, 2> t_range;
20  typedef t_tuple<t_vector, 2> t_plane;
21  typedef t_tuple<t_index, 2> t_edge;
22  typedef t_list<t_index> t_face;
23
24  struct t_body {
25    std::vector<t_vector> VERT;
26    std::vector<t_edge> EDGE;
27    std::vector<t_face> FACE;
28  };
29
30  t_vector operator%(const t_vector &lhs, const t_vector &rhs) { return
    ↪  t_vector{lhs[1] * rhs[2] - lhs[2] * rhs[1], lhs[2] * rhs[0] - lhs[0] * rhs[2],
    ↪  lhs[0] * rhs[1] - lhs[1] * rhs[0]}; }
31  t_vector operator-(const t_vector &lhs, const t_vector &rhs) { return
    ↪  t_vector{lhs[0] - rhs[0], lhs[1] - rhs[1], lhs[2] - rhs[2]}; }
32  t_vector operator+(const t_vector &lhs, const t_vector &rhs) { return
    ↪  t_vector{lhs[0] + rhs[0], lhs[1] + rhs[1], lhs[2] + rhs[2]}; }
33  t_scalar operator*(const t_vector &lhs, const t_vector &rhs) { return (lhs[0] *
    ↪  rhs[0] + lhs[1] * rhs[1] + lhs[2] * rhs[2]); }
34  t_vector operator*(const t_vector &vec, t_scalar val) { return t_vector{vec[0] *
    ↪  val, vec[1] * val, vec[2] * val}; }
35  t_vector operator*(t_scalar val, const t_vector &vec) { return vec * val; }
36  t_vector operator/(const t_vector &vec, t_scalar val) { return t_vector{vec[0] /
    ↪  val, vec[1] / val, vec[2] / val}; }
37  t_vector operator-(const t_vector &vec) { return -1.0 * vec; }
38  t_scalar len2(const t_vector &vec) { return (vec * vec); }
39  t_scalar len(const t_vector &vec) {
40    return std::sqrt(vec * vec);
41  }
42
43  constexpr t_scalar eps = 1.e-7;
44
45  t_scalar getVolume(const t_body &BODY) {
46    const auto &VERT = BODY.VERT;
47    const auto &EDGE = BODY.EDGE;
48    const auto &FACE = BODY.FACE;
49
50    std::vector<t_vector> FP(FACE.size()); t_vector p0 = {0., 0., 0.};
51    for (int f = 0; f < FACE.size(); ++ f) {
52      const auto &face = FACE[f];
53      auto &fp = FP[f]; fp = {0., 0., 0.};
54      for (int e: face) {
55        const auto &ed = EDGE[e];
56        auto &v1 = VERT[ed[0]];
57        auto &v2 = VERT[ed[1]];
58        fp = fp + v1 + v2;
59      }
60      fp = fp / (2. * face.size
61      p0 = p0 + fp;
62    }
63
64    p0 = p0 / FACE.size();
```

```
65
66    t_scalar vol = 0;
67    for (int f = 0; f < FACE.size(); ++ f) {
68      const auto &p = FP[f], &n = p - p0;
69      for (int e: FACE[f]) {
70        const t_edge &ed = EDGE[e];
71        auto &a = VERT[ed[0]];
72        auto &b = VERT[ed[1]];
73        auto c = (a - p) % (b - p);
74        auto d = len2(c);
75        if (d < eps * eps) continue;
76        d = std::sqrt(d);
77        auto h = std::abs(n * c) / d;
78        auto s = d / 2;
79        vol += h * s;
80      }
81    }
82
83    vol = vol / 3;
84    return vol;
85  }
86
87  t_range getRange(const t_body &BODY, const t_vector &top, const t_vector &dir) {
88    t_range R{
89      + std::numeric_limits<t_scalar>::max(),
90      - std::numeric_limits<t_scalar>::max()
91    };
92
93    for (const t_vector &v: BODY.VERT) {
94      const t_scalar s = (v - top) * dir;
95      R[0] = std::min(R[0], s);
96      R[1] = std::max(R[1], s);
97    }
98
99    return R;
100 }
101
102 t_body cutBody(const t_body &BODY, const t_vector &top, const t_vector &ort) {
103   enum t_state { CROSS, LOWER, UPPER, INNER };
104   typedef std::vector<int> t_child;
105   const auto &old_vert = BODY.VERT;
106   const auto &old_edge = BODY.EDGE;
107   const auto &old_face = BODY.FACE;
108
109   t_body NEW_BODY;
110   auto &new_vert = NEW_BODY.VERT;
111   auto &new_edge = NEW_BODY.EDGE;
112   auto &new_face = NEW_BODY.FACE;
113
114   std::vector<int> new_vert_index(old_vert.size(), -1);
115   std::vector<int> old_vert_state(old_vert.size());
116
117   for (int i = 0; i < old_vert.size(); ++ i) {
118     t_scalar s = (old_vert[i] - top) * ort;
119     if (std::abs(s) < eps) {
120       new_vert_index[i] = new_vert.size();
121       new_vert.push_back(old_vert[i]);
122       old_vert_state[i] = +0;
123     }
124     else
```

```
125       if (s < 0) {
126         new_vert_index[i] = new_vert.size();
127         new_vert.push_back(old_vert[i]);
128         old_vert_state[i] = -1;
129       }
130       else {
131         old_vert_state[i] = 1;
132       }
133     }
134
135     std::vector<int> new_edge_index(old_edge.size(), -1);
136     std::vector<t_state> old_edge_state(old_edge.size());
137     std::vector<t_child> new_edge_child(old_edge.size());
138
139     for (int i = 0; i < old_edge.size(); ++ i) {
140       const auto &edge = old_edge[i]; int a = edge[0], b = edge[1];
141
142       if (old_vert_state[a] == 0) {
143         new_edge_child[i].push_back(new_vert_index[a]);
144       }
145
146       if (old_vert_state[b] == 0) {
147         new_edge_child[i].push_back(new_vert_index[b]);
148       }
149
150       if (old_vert_state[a] * old_vert_state[b] < 0)
151         old_edge_state[i] = CROSS;
152       else
153       if (old_vert_state[a] + old_vert_state[b] < 0)
154         old_edge_state[i] = LOWER;
155       else
156       if (old_vert_state[a] + old_vert_state[b] > 0)
157         old_edge_state[i] = UPPER;
158       else {
159         old_edge_state[i] = INNER;
160       }
161
162       if (old_edge_state[i] == CROSS) {
163         const auto &va = old_vert[a], &vb = old_vert[b];
164         t_scalar sa = (va - top) * ort;
165         t_scalar sb = (vb - top) * ort;
166         t_vector ab = (vb - va);
167         auto p = - sa / (ab * ort);
168         auto v = va + ab * p;
169         new_edge_child[i].push_back(new_vert.size());
170
171         int ind = (old_vert_state[a] <= 0)? a: b;
172         new_edge_index[i] = new_edge.size();
173         new_edge.push_back({
174           new_vert_index[ind], int(new_vert.size())
175         });
176
177         new_vert.push_back(v);
178       }
179
180       if (old_edge_state[i] == INNER || old_edge_state[i] == LOWER) {
181         new_edge_index[i] = new_edge.size();
182         new_edge.push_back({
183           new_vert_index[a],
184           new_vert_index[b]
```

```
185         });
186         }
187     }
188 
189     std::vector<t_state> old_face_state(old_face.size());
190     std::vector<t_child> new_face_child(old_face.size());
191 
192     for (int i = 0; i < old_face.size(); ++ i) {
193       std::set<int> part;
194       t_face face;
195       int cross = 0, lower = 0, upper = 0, inner = 0;
196 
197       for (int e: old_face[i]) {
198         if (old_edge_state[e] == CROSS) ++ cross;
199         if (old_edge_state[e] == LOWER) ++ lower;
200         if (old_edge_state[e] == UPPER) ++ upper;
201         if (old_edge_state[e] == INNER) {
202           new_face_child[i].push_back(new_edge_index[e]);
203           ++ inner;
204         }
205 
206         if (old_edge_state[e] != UPPER) {
207           face.push_back(new_edge_index[e]);
208         }
209 
210         for (int k: new_edge_child[e]) {
211           part.insert(k);
212         }
213       }
214 
215       if (inner == old_face[i].size()) {
216         old_face_state[i] = INNER;
217       }
218       else
219       if (cross || (upper && lower)) {
220         if (inner || (part.size() != 2))
221           throw std::logic_error("ERROR: NOT CONVEX!");
222           old_face_state[i] = CROSS;
223       }
224       else {
225         old_face_state[i] = lower?
226         LOWER: UPPER;
227       }
228 
229       if (old_face_state[i] == CROSS) {
230         t_edge edge;
231         std::copy(part.begin(), part.end(), edge.begin());
232         new_face_child[i].push_back(new_edge.size());
233         //...
234         face.push_back(new_edge.size());
235         //...
236         new_edge.push_back(edge);
237       }
238 
239       if (old_face_state[i] != UPPER) {
240         if (face.size() < 3) {
241           throw std::logic_error("ERROR: SINGULAR!");
242         }
243 
244         new_face.push_back(face);
```

```
245       }
246     }
247
248     int cross = 0, lower = 0, upper = 0, inner = 0;
249
250     std::set<int> part;
251     for (int f = 0; f < old_face.size(); ++ f) {
252       if (old_face_state[f] == CROSS) ++ cross;
253       if (old_face_state[f] == LOWER) ++ lower;
254       if (old_face_state[f] == UPPER) ++ upper;
255       if (old_face_state[f] == INNER) ++ inner;
256       for (int k: new_face_child[f]) {
257         part.insert(k);
258       }
259     }
260
261     if (cross || (upper && lower)) {
262       new_face.push_back(t_face(part.begin(), part.end()));
263     }
264     return NEW_BODY;
265   }
266
267   //...
268
269   std::istream &operator>>(std::istream &inp, t_body &BODY) {
270     BODY.VERT.clear(); BODY.EDGE.clear(); BODY.FACE.clear();
271
272     std::map<t_edge, int> EDGE2IND;
273
274     auto &VERT = BODY.VERT;
275     auto &EDGE = BODY.EDGE;
276     auto &FACE = BODY.FACE;
277
278     int num_vert; inp >> num_vert;
279     for (int i = 0; i < num_vert; ++ i) {
280       VERT.push_back({}); for (auto &v: VERT.back()) inp >> v;
281     }
282     int num_face; inp >> num_face;
283
284     for (int i = 0; i < num_face; ++ i) {
285       int num; inp >> num;
286       int v0; inp >> v0;
287       int v1 = v0;
288       t_face face;
289       for (int k = 0; k < num; ++ k) {
290         int v2; if (k < num - 1) inp >> v2; else v2 = v0;
291         auto ed = (v1 < v2)? t_edge{v1, v2}: t_edge{v2, v1};
292         auto it = EDGE2IND.find(ed);
293         if (it == EDGE2IND.end()) {
294           int ind = EDGE.size();
295           face.push_back(ind);
296           EDGE2IND[ed] = ind;
297           EDGE.push_back(ed);
298         }
299         else {
300           face.push_back(it->second);
301         }
302         v1 = v2;
303       }
304
```

```
305       FACE.push_back(face);
306     }
307     return inp;
308   }
309
310   t_scalar getParam(const t_body &BODY, const t_range &range, const t_vector &ort,
      ↪  t_scalar vol) {
311     t_scalar par1 = range[0];
312     t_scalar vol1 = getVolume(cutBody(BODY, par1 * ort, ort));
313     t_scalar par2 = range[1];
314     t_scalar vol2 = getVolume(cutBody(BODY, par2 * ort, ort));
315     if ((vol1 > vol) || (vol2 < vol)) {
316       return std::numeric_limits<t_scalar>::quiet_NaN();
317     }
318     while ((std::abs(vol1 - vol) > eps) && (std::abs(vol2 - vol) > eps)) {
319       auto par0 = (par1 + par2) / 2.;
320       auto vol0 = getVolume(cutBody(BODY, par0 * ort, ort));
321       if (vol0 < vol) {
322         par1 = par0; vol1 = vol0;
323       }
324       else {
325         par2 = par0; vol2 = vol0;
326       }
327     }
328     if (std::abs(vol1 - vol) <= eps) {
329       return par1;
330     }
331     else {
332       return par2;
333     }
334   }
335
336   int main() {
337     std::ofstream fout("output.txt"); fout << std::setprecision(5) << std::fixed;
338     std::ifstream finp("input.txt");
339     t_body BODY; finp >> BODY; auto vol0 = getVolume(BODY);
340
341     const t_vector top0 = t_vector{0., 0., 0.};
342     t_vector ort1; finp >> ort1[0] >> ort1[1] >> ort1[2];
343     t_vector ort2; finp >> ort2[0] >> ort2[1] >> ort2[2];
344     ort1 = ort1 / len(ort1);
345     ort2 = ort2 / len(ort2);
346
347     for (int t = 0, dir = 1; t < 2; ++ t, dir *= -1) {
348       auto ran1 = getRange(BODY, top0, ort1);
349       auto ran2 = getRange(BODY, top0, ort2);
350
351       auto par1 = dir * getParam(BODY, ran1, dir * ort1, vol0 / 3.);
352
353       if (par1 != par1) {
354         if (t == 0) continue;
355         fout << "0";
356         return 0;
357       }
358
359       auto red1 = cutBody(BODY, par1 * ort1, - dir * ort1);
360       auto tmp1 = cutBody(BODY, par1 * ort1, dir * ort1);
361       auto ran0 = getRange(tmp1, top0, ort2);
362
363       ran2 = (std::abs(ran2[0] - ran0[0]) >
```

```
364             std::abs(ran0[1] - ran2[1]))?
365             t_range{ran2[0], ran0[0]}:
366             t_range{ran0[1], ran2[1]};
367
368     auto par2 = getParam(red1, ran2, ort2, vol0 / 3.);
369     if (par2 != par2) {
370       if (t == 0) continue;
371       fout << "0";
372       return 0;
373     }
374     auto red2 = cutBody(red1, par2 * ort2, -ort2);
375     auto tmp2 = cutBody(red1, par2 * ort2, ort2);
376     fout << "1\n";
377     fout << (par1 > 0? " ": "") << par1 << " ";
378     fout << (par2 > 0? " ": "") << par2;
379     fout << "\n";
380     break;
381   }
382   return 0;
383 }
```