Федеральное государственное автономное образовательное учреждение высшего образования
"Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского" (ННГУ)

Институт информационных технологий, математики и механики

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

«Построение выпуклой оболочки – проход Джарвиса»

Выполнил: студент группы 381706-1
Кольтюшкина Янина Вадимовна
_____ Подпись

Проверил:
Доцент кафедры МОСТ, кандидат технических наук
_____ Сысоев А. В.

Нижний Новгород
2019

Содержание

1. Введение

Пусть на плоскости задано конечное множество точек A. **Оболочкой** этого множества называется любая замкнутая линия H без самопересечений такая, что все точки из A лежат внутри этой кривой. Если кривая H является выпуклой (например, любая касательная к этой кривой не пересекает ее больше ни в одной точке), то соответствующая оболочка также называется **выпуклой**. Наконец, **минимальной выпуклой оболочкой** называется выпуклая оболочка минимальной длины (минимального периметра). [1]

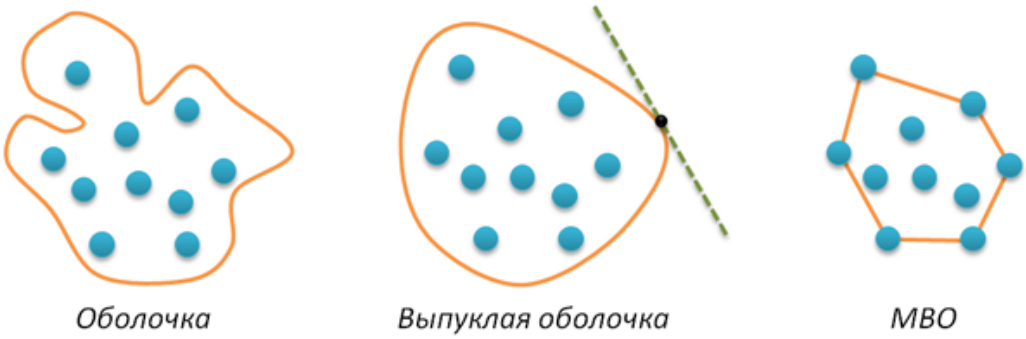

Рисунок 1 Отличия всех приведенных понятий [1]

Главной особенностью МВО множества точек A является то, что эта оболочка представляет собой выпуклый многоугольник, вершинами которого являются некоторые точки из A. Поэтому задача поиска МВО в конечном итоге сводится к отбору и упорядочиванию нужных точек из A. Упорядочивание является необходимым по той причине, что выходом алгоритма должен быть многоугольник, т.е. последовательность вершин. [1]

Цель – реализовать алгоритм прохода Джарвиса, используемый для составления выпуклой оболочки из массива точек на плоскости.

2. Постановка задачи

Для точек a_1, …, a_n, где n≥1, a_i=($a_{i,1}$, $a_{i,2}$) ∈ R^2 при i=1, …, n, указать вершины b_1, …, b_m выпуклой оболочки Conv(a_1, …, a_n) в порядке их встречи при движении по ее границе. Заметим, что в общем случае Conv(a_1, …, a_n) будет многоугольником, а в вырожденных случаях может получиться отрезок или точка. В случае отрезка выходом решающего поставленную задачу алгоритма должны быть две являющиеся его концами точки, а в случае точки - сама эта точка. [2] Помимо этого, алгоритм построения выпуклой оболочки необходимо распараллелить для корректной работы на разном числе процессов.

3. Описание алгоритмов

3.1. Построение выпуклой оболочки с помощью прохода Джарвиса

В качестве начальной берется самая левая нижняя точка среди всех $a_1, \ldots, a_n$. Путем простого прохода по множеству мы находим точку с минимальной первой координатой, а уже среди них ищем минимальную и вторую координату. Полученная точка $p_1 = \text{lexmin}(a_1, \ldots, a_n)$ является лексикографическим минимумом точек $a_1, \ldots, a_n$ и точно является вершиной выпуклой оболочки. Следующей точкой (p_2) берем такую, которая имеет наименьший положительный полярный угол относительно точки p_1 как начала координат. После этого для каждой точки p_i ($2 < i \leq |a|$) против часовой стрелки ищется такая точка p_{i+1}, путем нахождения среди оставшихся (+ первая), в которой будет образовываться наибольший угол между прямыми $p_{i-1}\ p_i$ и $p_i\ p_{i+1}$. Она и будет следующей вершиной выпуклой оболочки. Сам угол искать не обязательно, достаточно найти его косинус (через скалярное произведение между лучами $p_{i-1}\ p_i$ и $p_i\ p'_{i+1}$, где p'_{i+1} - претендент на следующий минимум. Новым минимумом будет та точка, в которой косинус будет принимать наименьшее значение. Ведь чем меньше косинус, тем больше его угол. Данный процесс заканчивается, когда следующая точка в выпуклой оболочке совпадает с первой. [3]

3.2.Быстрая сортировка

Быстрая сортировка относится к алгоритмам «разделяй и властвуй». Алгоритм состоит из трёх шагов:

1. Выбрать элемент из массива. Назовём его опорным.
2. Разбиение: перераспределение элементов в массиве таким образом, что элементы меньше опорного помещаются перед ним, а больше или равные после.
3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы. [4]

Можно по-разному выбирать опорный элемент, из-за чего скорость работы может отличаться. В данной работе выбран наиболее часто встречающийся вариант – середина текущего подмассива.

4. Схема распараллеливания

В данной работе параллельные вычисления организованы при подготовке к проходу Джарвиса (процессы вычисляют полярные координаты некоторого числа точек, если быть точным, то все точки/количество процессов), а также непосредственно в самом построении выпуклой оболочки. Рассмотрим этот процесс подробнее.

Среди всех точек ищется их лексикографический минимум – эта точка точно будет вершиной выпуклой оболочки. Далее определяется в каких четвертях будут лежать оставшиеся точки после параллельного переноса системы координат в найденный лексикографический минимум. Возможны два случая: все точки лежат в первой четверти или точки лежат в первой и четвертой четвертях. Далее мы разбиваем плоскость на зоны (их число = число процессов – 1). Это нужно нам для нахождения точек, которые с большей вероятностью попадут в итоговый результат. Мы определяем границы для полярного угла точек, попадающих в зоны: Если четвертей две, то 180/(число процессов – 1), если же четверть одна, то 90/(число процессов – 1). После чего, нулевой процесс ищет в каждой зоне точку с наибольшим полярным радиусом и рассылает их всем остальным процессам. Теперь нулевой процесс строит выпуклую оболочку из точек от 0 до найденной в первой зоне; первый из точек от найденной в первой зоне до найденной во второй + изначальная точка; и так далее. После чего, все результаты отправляются обратно к нулевому и тот проверяет правильность их построения.

Так же отмечу, что все построения выпуклых оболочек происходят согласно проходу Джарвиса, алгоритм которого описан выше.

5. Описание программной реализации

В программе используется несколько методов, необходимых для ее правильной работы. В первую очередь, это, конечно, построение выпуклой оболочки ConvexHull. Помимо нее, есть создание случайного массива точек заданной длины RandomHull, это полезно для проверки правильности работы и используется в тестах. О которых чуть позже. Функция cosvec вычисляет косинус угла через скалярное произведение, что является основой прохода Джарвиса. Методы PointMore и PointLess являются перегруженными операторами сравнения и используются в быстрой сортировки, которая нужна для упрощения вычислений. И остается сама быстрая сортировка QuickSort.

Из них всех распараллеливалась функция ConvexHull.

6. Подтверждение корректности

Чтобы проверить корректность работы программы мной были написаны 6 тестов. В них рассматриваются как обычные случаи (когда точки расположены в треугольнике или квадрате), при введенных мной входных данных (соответственно, результат, который должен получиться известен заранее), так и более нестандартные примеры с точками, расположенными на одной прямой. Есть тест, где точка просто одна. Но для достижения лучшего результата проверки работоспособности, был написан тест, в котором точки генерируются случайным образом, после чего выпуклая оболочка строится последовательным и параллельными способами, затем полученные результаты сравниваются друг с другом.

Тесты успешно работают, что и является подтверждением корректности.

7. Эксперименты

Эксперименты проводились на ПК с следующими параметрами:

1. Операционная система: Windows 10 Корпоративная 2016
2. Процессор: Intel(R) Core™ i7-4710MQ CPU @ 2.50 GHz
3. Оперативная память: 8 Gb
4. Версия Visual Studio: 2017

В таблице 1 приведена зависимость времени работы алгоритма при разном числе процессов.

Количество элементов = 500000.

Таблица 1. Время работы алгоритма в зависимости от числа процессов.

Количество процессов	Время работы последовательного алгоритма	Время работы параллельного алгоритма	Ускорение
2	4.83481	3.38436	1.428
3	4.70041	2.58523	1.818
4	4.82583	1.95033	2.474
6	5.02634	3.51844	1.429

Из таблицы видно, что программа работает эффективно и дает достаточно существенный прирост в скорости вычислений. При числе процессов <= 4 наблюдается правильная тенденция – чем больше процессов, тем быстрее происходят параллельные вычисления. Можно так же заметить, что при числе процессов больше 4 ускорение не столь существенное, это связано с параметрами процессора (он имеет всего 4 ядра).

8. Заключение

Благодаря данной работе, я смогла понять как в принципе строится выпуклая оболочка и как это лучше реализовать в виде программы. Мной был написан алгоритм построения выпуклой оболочки множества точек на плоскости с использованием прохода Джарвиса, который успешно работает как в последовательном случае (если запускать его на одном процессе), так и в параллельном. Созданные для проверки тесты подтверждают, что алгоритм не просто работает, а работает правильно.

9. Литература

1. Habr: Сообщество IT-специалистов: https://habr.com/ru/post/144921/
2. Груздев Д. В., Таланов В. А. Алгоритмы и структуры данных (лабораторные работы): [https://vk.com/doc51644906_515702336?hash=fc61c591076938632c&dl=0b9843bdc392de49ac], 2004.
3. Википедия: свободная электронная энциклопедия на русском языке: https://ru.wikipedia.org/wiki/Алгоритм_Джарвиса
4. Википедия: свободная электронная энциклопедия на русском языке: https://ru.wikipedia.org/wiki/Быстрая_сортировка

10. Приложение

10.1. convex_hull_jarvis.h

```
// Copyright 2019 Koltyushkina Yanina
#ifndef
MODULES_TASK_3_KOLTYUSHKINA_YA_CONVEX_HULL_JARVIS_CONVEX_HULL
_JARVIS_H_
#define
MODULES_TASK_3_KOLTYUSHKINA_YA_CONVEX_HULL_JARVIS_CONVEX_HULL
_JARVIS_H_


double** RandomHull(int size);

bool PointMore(int ind, double* mid, double* fi, double* r);
bool PointLess(int ind, double* mid, double* fi, double* r);

double** QuickSort(double** arr, int first, int last, double* fi, double* r);

double cosvec(double* p1, double* p2, double* p3);
double** ConvexHull(double** arr, const int nump);


#endif                                                                    //
MODULES_TASK_3_KOLTYUSHKINA_YA_CONVEX_HULL_JARVIS_CONVEX_HULL
_JARVIS_H_
```

10.2. convex_hull_jarvis.cpp

```
// Copyright 2019 Koltyushkina Yanina

#include <mpi.h>
#include <iostream>
#include <random>
#include <cmath>
#include <stdexcept>
#include "../../../modules/task_3/koltyushkina_ya_convex_hull_jarvis/convex_hull_jarvis.h"

double** RandomHull(int size) {
  if (size <= 0)
    throw "Negativ size";
  double** arr = new double*[size];
  for (int i = 0; i < size; i++)
    arr[i] = new double[2];
  std::mt19937 seed;
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < 2; j++) {
      arr[i][j] = seed() % 100;
    }
```

```
  }
  return arr;
}

bool PointMore(int ind, double* mid, double* fi, double* r) {
  if (fi[ind] > mid[0]) {
    return true;
  } else if (fi[ind] == mid[0]) {
    if (r[ind] > mid[1])
      return true;
  }
  return false;
}

bool PointLess(int ind, double* mid, double* fi, double* r) {
  if (fi[ind] < mid[0]) {
    return true;
  } else if (fi[ind] == mid[0]) {
    if (r[ind] < mid[1])
      return true;
  }
  return false;
}

double** QuickSort(double** arr, int first, int last, double* fi, double* r) {
  double* mid = new double[2];
  double temp;
  int f = first, l = last;
  mid[0] = fi[(f + l) / 2];
  mid[1] = r[(f + l) / 2];
  do {
    while (PointLess(f, mid, fi, r))
      f++;
    while (PointMore(l, mid, fi, r))
      l--;

    if (f <= l) {
      temp = r[f];
      r[f] = r[l];
      r[l] = temp;
      temp = fi[f];
      fi[f] = fi[l];
      fi[l] = temp;

      temp = arr[f][0];
      arr[f][0] = arr[l][0];
      arr[l][0] = temp;
      temp = arr[f][1];
      arr[f][1] = arr[l][1];
      arr[l][1] = temp;
```

```
      f++;
      l--;
    }
  } while (f < l);
  if (first < l)
    QuickSort(arr, first, l, fi, r);
  if (f < last)
    QuickSort(arr, f, last, fi, r);
  return arr;
}

double cosvec(double* p1, double* p2, double* p3) {
  double ax = p2[0] - p1[0];
  double ay = p2[1] - p1[1];
  double bx = p3[0] - p1[0];
  double by = p3[1] - p1[1];
  return ((ax*bx + ay * by) / (sqrt(ax*ax + ay * ay) * sqrt(bx*bx + by * by)));
}

double** ConvexHull(double** arr, const int nump) {
  double** nowres = new double*[nump + 1];
  double** result = new double*[nump + 1];
  for (int i = 0; i < nump + 1; i++) {
    nowres[i] = new double[2];
    result[i] = new double[2];
  }
  nowres[0][0] = 0;
  if (nump == 1) {
    result[0][0] = 1;
    result[1] = arr[0];
    return result;
  } else if (nump == 2) {
    result[0][0] = 2;
    result[1] = arr[0];
    result[2] = arr[1];
  } else {
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int delta;
    int ost;
    int f = 0;
    ost = nump % size;
    if ((nump < size) || (nump / size <= 2)) {
      ost = 0;
      f = -1;
      size = 1;
    }
    delta = nump / size;

    double* OO = new double[2];
```

```
OO[0] = arr[0][0];
OO[1] = arr[0][1];
int m = 0;
for (int i = 1; i < nump; i++) {
  if (arr[i][0] < OO[0]) {
    OO[0] = arr[i][0];
    OO[1] = arr[i][1];
    m = i;
  } else {
    if (arr[i][0] == OO[0]) {
      if (arr[i][1] < OO[1]) {
        OO[0] = arr[i][0];
        OO[1] = arr[i][1];
        m = i;
      }
    }
  }
}

double* r = new double[nump];
double* fi = new double[nump];
double* tmp = arr[0];
arr[0] = arr[m];
arr[m] = tmp;
m = 0;
for (int i = 1; i < nump; i++)
  for (int j = 0; j < 2; j++)
    arr[i][j] = arr[i][j] - OO[j];

if (rank == 0) {
  for (int i = 1; i < delta + ost; i++) {
    r[i] = pow((arr[i][0] * arr[i][0]) + (arr[i][1] * arr[i][1]), 0.5);
    fi[i] = atan(arr[i][1] / arr[i][0]);
  }
} else {
  if (f == 0) {
    for (int i = delta * rank + ost; i < delta*rank + delta + ost; i++) {
      r[i] = pow((arr[i][0] * arr[i][0]) + (arr[i][1] * arr[i][1]), 0.5);
      fi[i] = atan(arr[i][1] / arr[i][0]);
    }
  }
}
if (size > 1) {
  MPI_Bcast(&r[1], delta + ost - 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&fi[1], delta + ost - 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
for (int i = 1; i < size; i++) {
  MPI_Bcast(&fi[delta * i + ost], delta, MPI_DOUBLE, i, MPI_COMM_WORLD);
  MPI_Bcast(&r[delta*i + ost], delta, MPI_DOUBLE, i, MPI_COMM_WORLD);
}
arr[0][0] = 0;
```

```
arr[0][1] = 0;

int lockind;

MPI_Barrier(MPI_COMM_WORLD);
QuickSort(arr, 1, nump - 1, fi, r);
int ind;
if (rank == 0) {
  int smind = 1;
  if (fi[1] == fi[nump - 1]) {
    smind = nump - 1;
  } else if (fi[1] == fi[2]) {
    smind = 2;
    while (fi[smind] == fi[smind + 1]) {
      smind += 1;
    }
  }
  ind = smind;
}

MPI_Bcast(&ind, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (ind == nump - 1) {
  result[0][0] = 2;
  result[1][0] = arr[0][0] + OO[0];
  result[1][1] = arr[0][1] + OO[1];
  result[2][0] = arr[nump - 1][0] + OO[0];
  result[2][1] = arr[nump - 1][1] + OO[1];
  return result;
}
int pcount = nump;
int minind = ind;
if (rank == 0) {
  int smind = nump - 1;
  int endind = nump;
  while (fi[smind] == fi[smind - 1]) {
    smind--;
  }
  if (smind != nump - 1) {
    endind = smind + 1;
  }
  pcount = endind;
}
MPI_Bcast(&pcount, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (f == -1) {
  ost = 0;
  size = 1;
  arr[pcount - 1][0] = arr[nump - 1][0];
  arr[pcount - 1][1] = arr[nump - 1][1];
  nowres[1] = arr[0];
```

```
  nowres[2] = arr[ind];
  nowres[0][0] = 2;
  delta = (pcount - ind - 1);
  int sizeres = 2;
  int flag = 0;
  while (flag != 1) {
    double* last = new double[2];
    double* beforelast = new double[2];
    last = nowres[sizeres];
    beforelast = nowres[sizeres - 1];
    int minind = 0;
    double mincos = cosvec(last, beforelast, arr[0]);
    double cos0 = mincos;
    double nowcos;

    for (int j = ind + 1; j < pcount; j++) {
      nowcos = cosvec(last, beforelast, arr[j]);
      if ((nowcos <= mincos) && (nowcos != cos0)) {
        mincos = nowcos;
        minind = j;
      }
    }
    if (minind == 0) {
      flag = 1;
      break;
    } else {
      sizeres++;
      nowres[sizeres] = arr[minind];
      delta = (pcount - minind - 1) / size;
      ost = (pcount - minind - 1) % size;
      ind = minind;
    }
  }
  result[0][0] = sizeres; for (int i = 0; i < sizeres; i++) {
    result[i + 1][0] = nowres[i + 1][0] + OO[0];
    result[i + 1][1] = nowres[i + 1][1] + OO[1];
  }
  return result;
}
arr[pcount - 1][0] = arr[nump - 1][0];
arr[pcount - 1][1] = arr[nump - 1][1];

nowres[1] = (arr[0]);
nowres[2] = (arr[ind]);
int sizeres = 2;

delta = (pcount - ind - 1) / size;
ost = (pcount - ind - 1) % size;
double* arrmincos = new double[size];
int* arrminind = new int[size];
```

```
    double* last = new double[2];
    double* beforelast = new double[2];

    while (delta > 1) {
      last = nowres[sizeres];
      beforelast = nowres[sizeres - 1];
      minind = 0;
      double mincos = cosvec(last, beforelast, arr[0]);
      double cos0 = mincos;
      double nowcos;
      if (rank == 0) {
        int endlockind;
        if (ind + delta + ost + 1 > nump) {
          endlockind = nump;
        } else {
          endlockind = ind + delta + ost + 1;
          for (int j = ind + 1; j < endlockind; j++) {
            nowcos = cosvec(last, beforelast, arr[j]);
            if ((nowcos <= mincos) && (nowcos != cos0)) {
              mincos = nowcos;
              minind = j;
            }
          }
        }
      } else {
        if (f == 0) {
          lockind = ind + delta * rank + ost + 1;
          for (int j = lockind; j < lockind + delta; j++) {
            if (j == lockind) {
              mincos = cosvec(last, beforelast, arr[j]);
            } else {
              nowcos = cosvec(last, beforelast, arr[j]);
              if ((nowcos <= mincos) && (nowcos != cos0)) {
                mincos = nowcos;
                minind = j;
              }
            }
          }
        }
      }

      MPI_Gather(&minind, 1, MPI_INT, arrminind, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Gather(&mincos, 1, MPI_DOUBLE, arrmincos, 1, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

      if (rank == 0) {
        minind = arrminind[0];
        mincos = arrmincos[0];
        for (int i = 0; i < size; i++) {
          if ((arrmincos[i] <= mincos) && (arrmincos[i] != cos0)) {
            mincos = arrmincos[i];
```

```
          minind = arrminind[i];
        }
      }
    }

    MPI_Bcast(&minind, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    ind = minind;
    sizeres++;
    nowres[sizeres] = arr[minind];
    delta = (pcount - minind - 1) / size;
    ost = (pcount - minind - 1) % size;
  }
  if (rank == 0) {
    int flag = 0;
    while (flag != 1) {
      double* last = new double[2];
      double* beforelast = new double[2];
      last = nowres[sizeres];
      beforelast = nowres[sizeres - 1];
      minind = 0;
      double mincos = cosvec(last, beforelast, arr[0]);
      double cos0 = mincos;
      double nowcos;

      for (int j = ind + 1; j < pcount; j++) {
        nowcos = cosvec(last, beforelast, arr[j]);
        if ((nowcos <= mincos) && (nowcos != cos0)) {
          mincos = nowcos;
          minind = j;
        }
      }
      if (minind == 0) {
        flag = 1;
        break;
      } else {
        sizeres++;
        nowres[sizeres] = arr[minind];
        delta = (pcount - minind - 1) / size;
        ost = (pcount - minind - 1) % size;
        ind = minind;
      }
    }
    result[0][0] = sizeres;
    for (int i = 0; i < sizeres; i++) {
      result[i + 1][0] = nowres[i + 1][0] + OO[0];
      result[i + 1][1] = nowres[i + 1][1] + OO[1];
    }
  }
}
return result;
```

```
}
```

10.3. main.cpp

```
// Copyright 2019 Koltyushkina Yanina

#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <math.h>
#include <random>
#include "../../../modules/task_3/koltyushkina_ya_convex_hull_jarvis/convex_hull_jarvis.h"

TEST(Convex_Hull_Jarvis_mpi, test_one_point) {
  double** mas = new double*[1];
  double** result = new double*[2];
  double** myres = new double*[2];
  for (int i = 0; i < 1; i++) {
    mas[i] = new double[2];
  }
  for (int i = 0; i < 2; i++) {
    myres[i] = new double[2];
    result[i] = new double[2];
  }
  mas[0][0] = 0;
  mas[0][1] = 0;

  myres[1][0] = 0;
  myres[1][1] = 0;
  result = ConvexHull(mas, 1);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    for (int i = 1; i < 2; i++) {
      for (int j = 0; j < 2; j++)
        EXPECT_EQ(result[i][j], myres[i][j]);
    }
  }
}

TEST(Convex_Hull_Jarvis_mpi, test_triangle_point) {
  double** mas = new double*[4];
  for (int i = 0; i < 4; i++)
    mas[i] = new double[2];
  mas[0][0] = 0;
  mas[0][1] = 0;
  mas[1][0] = 2;
  mas[1][1] = 0;
  mas[2][0] = 2;
  mas[2][1] = 1;
  mas[3][0] = 2;
  mas[3][1] = 2;
```

```
  double** result = new double*[5];
  double** myres = new double*[5];
  for (int i = 0; i < 5; i++) {
    myres[i] = new double[2];
    result[i] = new double[2];
  }
  myres[1][0] = 0;
  myres[1][1] = 0;
  myres[2][0] = 2;
  myres[2][1] = 0;
  myres[3][0] = 2;
  myres[3][1] = 2;

  result = ConvexHull(mas, 4);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int s = static_cast<int>(result[0][0]);
    for (int i = 1; i < s + 1; i++) {
      for (int j = 0; j < 2; j++) {
        EXPECT_EQ(myres[i][j], result[i][j]);
      }
    }
  }
}

TEST(Convex_Hull_Jarvis_mpi, test_triangle_points) {
  double** mas = new double*[6];
  for (int i = 0; i < 6; i++)
    mas[i] = new double[2];
  mas[0][0] = 0;
  mas[0][1] = 0;
  mas[1][0] = 2;
  mas[1][1] = 0;
  mas[2][0] = 1;
  mas[2][1] = 1;
  mas[3][0] = 2;
  mas[3][1] = 2;
  mas[4][0] = 2;
  mas[4][1] = 1;
  mas[5][0] = 2;
  mas[5][1] = 1.75;

  double** result = new double*[7];
  double** myres = new double*[7];
  for (int i = 0; i < 7; i++) {
    myres[i] = new double[2];
    result[i] = new double[2];
  }
  myres[1][0] = 0;
```

```
  myres[1][1] = 0;
  myres[2][0] = 2;
  myres[2][1] = 0;
  myres[3][0] = 2;
  myres[3][1] = 2;

  result = ConvexHull(mas, 6);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int s = static_cast<int>(result[0][0]);
    for (int i = 1; i < s + 1; i++) {
      for (int j = 0; j < 2; j++) {
        EXPECT_EQ(myres[i][j], result[i][j]);
      }
    }
  }
}

TEST(Convex_Hull_Jarvis_mpi, test_square_points) {
  double** mas = new double*[7];
  for (int i = 0; i < 7; i++)
    mas[i] = new double[2];
  mas[0][0] = 0;
  mas[0][1] = 0;
  mas[1][0] = 1;
  mas[1][1] = 0;
  mas[2][0] = 1;
  mas[2][1] = 1;
  mas[3][0] = 0;
  mas[3][1] = 1;
  mas[4][0] = 0.25;
  mas[4][1] = 0;
  mas[5][0] = 0;
  mas[5][1] = 0.75;
  mas[6][0] = 0.75;
  mas[6][1] = 1;

  double** result = new double*[8];
  double** myres = new double*[8];
  for (int i = 0; i < 7; i++) {
    myres[i] = new double[2];
    result[i] = new double[2];
  }
  myres[1][0] = 0;
  myres[1][1] = 0;
  myres[2][0] = 1;
  myres[2][1] = 0;
  myres[3][0] = 1;
  myres[3][1] = 1;
```

```
  myres[4][0] = 0;
  myres[4][1] = 1;
  result = ConvexHull(mas, 7);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    int s = static_cast<int>(result[0][0]);
    for (int i = 1; i < s + 1; i++) {
      for (int j = 0; j < 2; j++) {
        EXPECT_EQ(myres[i][j], result[i][j]);
      }
    }
  }
}

TEST(Convex_Hull_Jarvis_mpi, test_straight) {
  double** mas = new double*[10];
  for (int i = 0; i < 10; i++)
    mas[i] = new double[2];
  for (int i = -2; i < 8; i++) {
    mas[i + 2][0] = i;
    mas[i + 2][1] = i;
  }


  double** result = new double*[11];
  double** myres = new double*[11];
  for (int i = 0; i < 11; i++) {
    myres[i] = new double[2];
    result[i] = new double[2];
  }
  myres[1][0] = -2;
  myres[1][1] = -2;
  myres[2][0] = 7;
  myres[2][1] = 7;

  result = ConvexHull(mas, 10);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int s = static_cast<int>(result[0][0]);
    for (int i = 1; i < s + 1; i++) {
      for (int j = 0; j < 2; j++) {
        EXPECT_EQ(myres[i][j], result[i][j]);
      }
    }
  }
}

TEST(Convex_Hull_Jarvis_mpi, test_straight_point) {
```

```
  double** mas = new double*[11];
  for (int i = 0; i < 11; i++)
    mas[i] = new double[2];
  for (int i = -2; i < 8; i++) {
    mas[i + 2][0] = i;
    mas[i + 2][1] = i;
  }
  mas[10][0] = 4;
  mas[10][1] = 7;

  double** result = new double*[12];
  double** myres = new double*[12];
  for (int i = 0; i < 12; i++) {
    myres[i] = new double[2];
    result[i] = new double[2];
  }
  myres[1][0] = -2;
  myres[1][1] = -2;
  myres[2][0] = 7;
  myres[2][1] = 7;
  myres[3][0] = 4;
  myres[3][1] = 7;



  result = ConvexHull(mas, 11);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    int s = static_cast<int>(result[0][0]);
    for (int i = 1; i < s + 1; i++) {
      for (int j = 0; j < 2; j++) {
        EXPECT_EQ(myres[i][j], result[i][j]);
      }
    }
  }
}

TEST(Convex_Hull_Jarvis_mpi, test_random_points) {
  int n = 100;
  double** mas = new double*[n];
  for (int i = 0; i < n; i++)
    mas[i] = new double[2];
  mas = RandomHull(n);
  double** newmas = new double*[n];
  for (int i = 0; i < n; i++) {
    newmas[i] = new double[2];
    for (int j = 0; j < 2; j++)
      newmas[i][j] = mas[i][j];
  }
```

```
double* tmp = NULL;
double** result = new double*[n];
double** myres = new double*[n + 1];
for (int i = 0; i < n + 1; i++) {
 myres[i] = new double[2];
 result[i] = new double[2];
}
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Barrier(MPI_COMM_WORLD);
if (rank == 0) {
 int delta = n;
 double* OO = new double[2];
 OO[0] = mas[0][0];
 OO[1] = mas[0][1];
 int m = 0;
 for (int i = 1; i < n; i++) {
  if (mas[i][0] < OO[0]) {
   OO[0] = mas[i][0];
   OO[1] = mas[i][1];
   m = i;
  }
  else {
   if (mas[i][0] == OO[0]) {
    if (mas[i][1] < OO[1]) {
     OO[0] = mas[i][0];
     OO[1] = mas[i][1];
     m = i;
    }
   }
  }
 }

 double* r = new double[n];
 double* fi = new double[n];
 tmp = mas[0];
 mas[0] = mas[m];
 mas[m] = tmp;
 m = 0;
 for (int i = 1; i < n; i++)
  for (int j = 0; j < 2; j++)
   mas[i][j] = mas[i][j] - OO[j];


 for (int i = 1; i < n; i++) {
  r[i] = pow((mas[i][0] * mas[i][0]) + (mas[i][1] * mas[i][1]), 0.5);
  fi[i] = atan(mas[i][1] / mas[i][0]);
 }

 mas[0][0] = 0;
```

```
mas[0][1] = 0;

QuickSort(mas, 1, n - 1, fi, r);
int ind;

int smind = 1;
if (fi[1] == fi[n - 1]) {
  smind = n - 1;
}
else if (fi[1] == fi[2]) {
  smind = 2;
  while (fi[smind] == fi[smind + 1]) {
    smind += 1;
  }
}
ind = smind;

if (ind == n - 1) {
  myres[0][0] = 2;
  myres[1][0] = mas[0][0] + OO[0];
  myres[1][1] = mas[0][1] + OO[1];
  myres[2][0] = mas[n - 1][0] + OO[0];
  myres[2][1] = mas[n - 1][1] + OO[1];
}
else {
  int pcount = n;
  int minind = ind;
  int smind = n - 1;
  int endind = n;
  while (fi[smind] == fi[smind - 1]) {
    smind--;
  }
  if (smind != n - 1) {
    endind = smind + 1;
  }
  pcount = endind;

  double** nowres = new double*[n + 1];
  for (int i = 0; i < n + 1; i++)
    nowres[i] = new double[2];
  nowres[0][0] = 0;

  mas[pcount - 1][0] = mas[n - 1][0];
  mas[pcount - 1][1] = mas[n - 1][1];
  nowres[1] = mas[0];
  nowres[2] = mas[ind];
  nowres[0][0] = 2;
  delta = (pcount - ind - 1);
  int sizeres = 2;
  int flag = 0;
  while (flag != 1) {
```

```
      double* last = new double[2];
      double* beforelast = new double[2];
      last = nowres[sizeres];
      beforelast = nowres[sizeres - 1];
      int minind = 0;
      double mincos = cosvec(last, beforelast, mas[0]);
      double cos0 = mincos;
      double nowcos;

      for (int j = ind + 1; j < pcount; j++) {
        nowcos = cosvec(last, beforelast, mas[j]);
        if ((nowcos <= mincos) && (nowcos != cos0)) {
          mincos = nowcos;
          minind = j;
        }
      }
      if (minind == 0) {
        flag = 1;
        break;
      }
      else {
        sizeres++;
        nowres[sizeres] = mas[minind];
        delta = (pcount - minind - 1);
        ind = minind;
      }
    }
    myres[0][0] = sizeres;
    for (int i = 0; i < sizeres; i++) {
      myres[i + 1][0] = nowres[i + 1][0] + OO[0];
      myres[i + 1][1] = nowres[i + 1][1] + OO[1];
    }
  }
 }

 MPI_Barrier(MPI_COMM_WORLD);
 result = ConvexHull(newmas, n);

 if (rank == 0) {
  int s = static_cast<int>(result[0][0]);
  for (int i = 1; i < s + 1; i++) {
    for (int j = 0; j < 2; j++) {
      EXPECT_EQ(myres[i][j], result[i][j]);
    }
  }
 }
}

int main(int argc, char** argv) {
 ::testing::InitGoogleTest(&argc, argv);
 MPI_Init(&argc, &argv);
```

```
  ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
  ::testing::TestEventListeners& listeners =
    ::testing::UnitTest::GetInstance()->listeners();

  listeners.Release(listeners.default_result_printer());
  listeners.Release(listeners.default_xml_generator());

  listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);

  return RUN_ALL_TESTS();
}
```