sid.inpe.br/mtc-m21b/2015/06.02.18.30-MAN

DESCRIPTION OF A C++ IMPLEMENTATION OF THE INCREMENTAL ALGORITHM TO GENERATE THE DELAUNAY TRIANGULATION

Henrique Rennó de Azeredo Freitas

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3JJUR2L>

INPE
São José dos Campos
2015

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
Fax: (012) 3208-6919
E-mail: pubtc@sid.inpe.br

sid.inpe.br/mtc-m21b/2015/06.02.18.30-MAN

DESCRIPTION OF A C++ IMPLEMENTATION OF THE INCREMENTAL ALGORITHM TO GENERATE THE DELAUNAY TRIANGULATION

Henrique Rennó de Azeredo Freitas

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3JJUR2L>

INPE
São José dos Campos
2015

ABSTRACT

This work describes the data structures and procedures developed as a C++ implementation of the Incremental algorithm to calculate a triangulation called Delaunay triangulation, which is often used in several scientific applications as, for example, in the modeling of terrain surfaces. The implementation includes functions that perform computational geometry tasks on a set of points used as input and then generates a set of triangles as output.

DESCRIÇÃO DE UMA IMPLEMENTAÇÃO EM C++ DO ALGORITMO INCREMENTAL PARA GERAR A TRIANGULAÇÃO DE DELAUNAY

RESUMO

Este trabalho descreve as estruturas de dados e os procedimentos desenvolvidos como uma implementação em C++ do algoritmo Incremental para calcular uma triangulação chamada triangulação de Delaunay, a qual é muito utilizada em diversas aplicações científicas como, por exemplo, na modelagem de superfícies de terreno. A implementação inclui funções que executam tarefas de geometria computacional em um conjunto de pontos utilizados como entrada e gera um conjunto de triângulos como saída.

LIST OF FIGURES

Page

CONTENTS

1 INTRODUCTION

Triangulations are structures generally comprised of several adjacent triangles of different sizes and shapes which are usually generated from a set of points distributed over space, where each point can be defined by a triple (x, y, z) with x and y as the coordinates on the horizontal plane and z as a particular value associated to the point (x, y).

These structures have relevant and useful applications in many areas of natural sciences since digital elevation models are efficiently represented by triangulations (PETRIE; KENNIE, 1987; JONES et al., 1990; BERG et al., 2008) where the density of information can vary from region to region, thus avoiding data redundancy and producing a model that better adapts to the irregularities of the terrain.

The present work describes all the data structures and procedures developed as a C++ implementation of the Incremental algorithm to generate a particular triangulation called Delaunay triangulation.

2 DELAUNAY TRIANGULATION

A triangulation calculated from a set of points in the plane is defined as a subdivision of the plane such that any three points of the set are connected by edges to form the vertices of a triangle and all the edges of the triangulation do not intersect each other at points different from the vertices. Many distinct triangulations can be calculated from the same set of points and one particular triangulation is called the Delaunay triangulation (BERG et al., 2008), which avoids the generation of skinny triangles.

The main property of the Delaunay triangulation is that each triangle defines a circle through its three vertices that does not contain any other point of the set in its interior, so that this property is considered as a criteria for calculating the triangulation (TSAI, 1993). As a consequence of this property, the Delaunay triangulation presents more equiangular triangles and the minimum angle among all the triangles is maximized. Figure 2.1 shows a Delaunay triangulation together with the Delaunay criteria for a circle defined by the three vertices of a triangle.

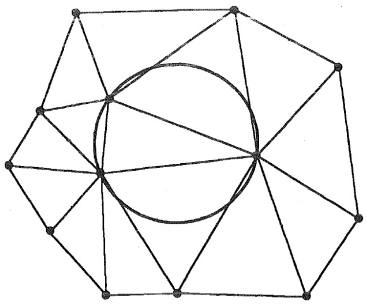

Figure 2.1 - Delaunay triangulation criteria
SOURCE: adapted from Jones et al. (1990)

The Delaunay triangulation can be generated by several different algorithms found in the literature, each one with a specific approach. Some of these algorithms are: Bowyer-Watson (BOWYER, 1981; WATSON, 1981), Incremental (GUIBAS et al., 1992; BERG et al., 2008), Divide-and-Conquer (CIGNONI et al., 1998), Fortune (FORTUNE, 1987) and Brute Force (O'ROURKE, 1998). This work describes the Incremental algorithm, which presents a time complexity of $O(n\ log\ n)$ (BERG et al., 2008) as an upper bound for the running time of the algorithm with an input dataset of size n. A C++ implementation of the algorithm is also given, with the data structures for points and triangles listed in annex A and all the global variables listed in annex B.

3 INCREMENTAL ALGORITHM

The first step of the Incremental algorithm is to define a triangle that contains the whole set of points to be triangulated located in its interior. In addition, each one of its vertices cannot be inside the circle that passes through any three points of the set, thus respecting the Delaunay criteria.

The coordinates of the vertices of this initial triangle are defined by a constant value K calculated from a point P that defines the width and height of an enclosing rectangle of the set of points as presented by Vomacka (2008) apud Zalik and Kolingerova (2003), where these authors consider the value of K as ten times greater than the larger rectangle dimension, either width or height.

This work considers the value of K calculated from a straight line with a slope equals to -1, where this line is translated up and to the right in relation to the point P, so that the intersections between the line and both the x and y axes determine the points $(K, 0)$ and $(0, K)$, respectively. The initial enclosing triangle is then defined by the vertices $(K, 0)$, $(0, K)$ and $(-K, -K)$ as illustrated in figure 3.1, where the black rectangle represents the enclosing rectangle of the set of points. The source code that defines the enclosing triangle is in annex C.

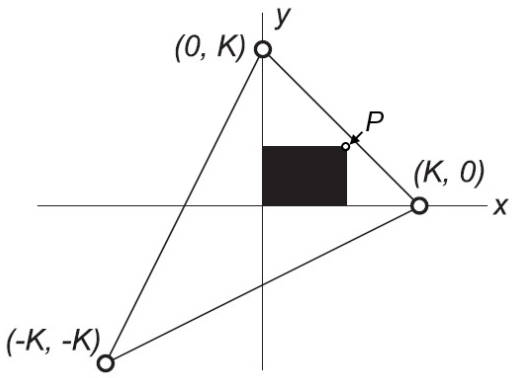

Figure 3.1 - Enclosing triangle of the set of points
SOURCE: adapted from Vomacka (2008)

After defining the enclosing triangle, each point of the set is inserted one at a time into the current triangulation, then new edges are created and connected to the inserted point. These new edges are either connected to each one of the three vertices of the triangle that contains the point, if the point is located inside the triangle, or to

the vertices opposite to an edge shared by two triangles, if the point is exactly on an edge. The new edges of the triangles created are then considered as Delaunay edges, i.e., these edges respect the Delaunay criteria since the circle that passes through them does not include any other point in its interior.

However, the edges of each divided triangle need to be checked whether they are still Delaunay edges as a circle passing through both the points of one of the edges and also through the inserted point may include another point of the set inside it. In this case, the edge will be considered illegal. Figure 3.2 exhibits an example of illegal edge, where the circle that passes through the points of the edge $p_i p_j$ and also through the point p_k includes the point p_l in its interior.

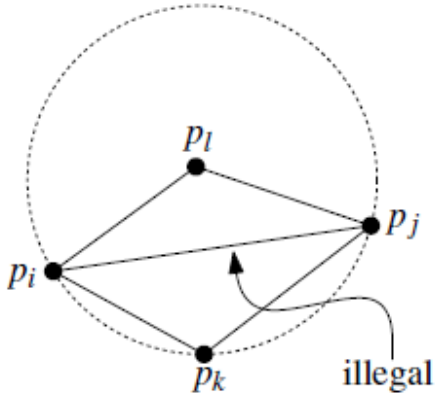

Figure 3.2 - Edge $p_i p_j$ is illegal
SOURCE: Berg et al. (2008)

Furthermore, when a triangle edge is considered illegal, it is modified by a rotation operation and its new points are the points opposite to the edge before the rotation, so that both the triangles that shared the edge before it is rotated are also changed into two new triangles. The rotated edge becomes a Delaunay edge since the interior angles of the new triangles is maximized as shown in figure 3.3, where two skinny triangles are changed after the rotation of the edge $p_i p_j$ to $p_k p_l$.

An edge rotation can make the other two edges of the triangles that existed before the rotation become illegal. These edges also need to be checked if they are still Delaunay edges, otherwise they are rotated as well. This procedure of checking and rotating recursively repeats until no more edges are illegal. Therefore the triangulation is locally changed and triangles are modified around the inserted point.

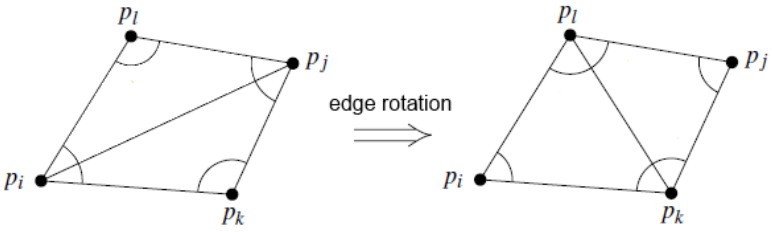

Figure 3.3 - Rotating an edge maximizes interior angles of triangles
SOURCE: adapted from Berg et al. (2008)

In the programming level, for the previously mentioned modifications to be correctly applied to the triangulation, the triangle structure holds the information of each adjacent triangle as a pointer reference and these references change with each edge rotation in order to ensure that the adjacency relationships are maintained.

The Incremental algorithm uses a tree structure for the storage of the triangles as its nodes, where divided triangles and new triangles created are then connected in the tree by hierarchical links that indicate the triangles divisions. The new triangles are stored as children of the divided triangles, so that each triangle has three or two children depending on whether the inserted point is located inside a triangle or on an edge, respectively. At the end, Delaunay triangles are leaf nodes of the tree and the enclosing triangle, together with all its incident edges, is discarded. An example of how the triangles and the tree structure are modified after a point is inserted can be visualized in figure 3.4. The source code of the Incremental algorithm that calculates the Delaunay triangulation is in annex D with specific geometric and utility functions in annexes E and F, respectively.

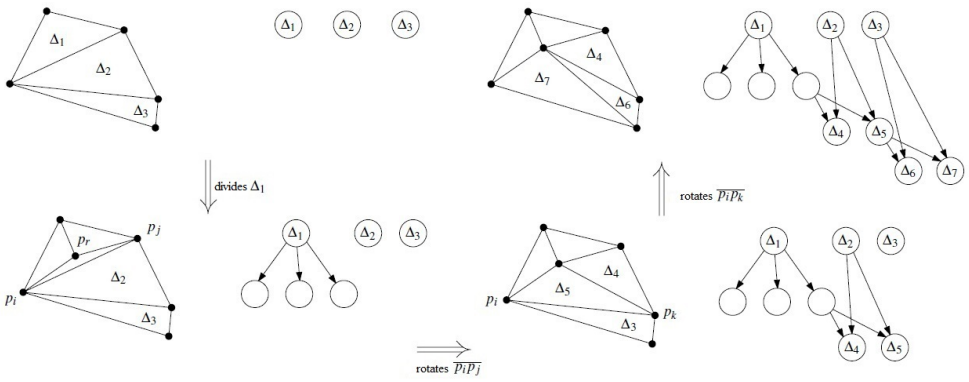

Figure 3.4 - Tree structure reflects triangles divisions
SOURCE: adapted from Berg et al. (2008)

4 CONCLUSIONS

The implementation in a programming language of the Incremental algorithm to generate the Delaunay triangulation is not a trivial problem. Despite being comprised of simple recursive procedures, there are many details related to the information that is stored in the data structures and also to particular tasks of computational geometry that must be solved for the algorithm to work properly.

This work presented the source code of the Incremental algorithm written in the C++ programming language with data structures and procedures developed as a set of several functions that can be used in a computational geometry library as an alternative implementation to generate the Delaunay triangulation.

REFERENCES

BERG, M. de; CHEONG, O.; KREVELD, M. v.; OVERMARS, M. **Computational geometry**: algorithms and applications. 3. ed. Santa Clara, CA, USA: Springer-Verlag, 2008. 191-205 p. ISBN 978-3-540-77973-5. 1, 3, 6, 7

BOWYER, A. Computing Dirichlet tessellations. **The Computer Journal**, v. 24, n. 2, p. 162–166, 1981. 3

CIGNONI, P.; MONTANI, C.; SCOPIGNO, R. DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed. **Computer-Aided Design**, v. 30, n. 5, p. 333–341, Apr 1998. 3

FORTUNE, S. A sweepline algorithm for Voronoi diagrams. **Algorithmica**, v. 2, p. 153–174, 1987. 3

GUIBAS, L. J.; KNUTH, D. E.; SHARIR, M. Randomized incremental construction of Delaunay and Voronoi diagrams. **Algorithmica**, v. 7, n. 1-6, p. 381–413, Jun 1992. 3

JONES, N. L.; WRIGHT, S. G.; MAIDMENT, D. R. Watershed delineation with triangle-based terrain models. **Journal of Hydraulic Engineering - ASCE**, v. 116, n. 10, p. 1232–1251, Oct 1990. 1, 3

O'ROURKE, J. **Computational geometry in C**. 2. ed. New York, NY, USA: Cambridge University Press, 1998. ISBN 0521640105. Available from: <http://cs.smith.edu/~orourke/books/compgeom.html>. 3

PETRIE, G.; KENNIE, T. J. M. Terrain modelling in surveying and civil engineering. **Computer-Aided Design**, Butterworth-Heinemann, Newton, MA, USA, v. 19, n. 4, p. 171–187, May 1987. ISSN 0010-4485. 1

TSAI, V. J. D. Delaunay triangulations in TIN creation: an overview and a linear-time algorithm. **International Journal of Geographical Information Systems**, v. 7, n. 6, p. 501–524, 1993. 3

VOMACKA, T. **Delaunay triangulation of moving points in a plane**. 2008. University of West Bohemia, Pilsen. 5

WATSON, D. F. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. **The Computer Journal**, v. 24, n. 2, p. 167–172, 1981. 3

ZALIK, B.; KOLINGEROVA, I. An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. **International Journal of Geographical Information Science**, v. 17, n. 2, p. 119–138, 2003. 5

ANNEX A - DATA STRUCTURES

```
struct POINT
{
    // point coordinates
    double x, y, z;
    // point memory address
    struct POINT *address;
    // point index in vector
    int point_index;
    // flags
    bool processed, visited;
};

struct TRIANGLE
{
    // points of the triangle
    struct POINT p_i, p_j, p_k;
    // number of children and parents of the triangle
    int number_of_children, number_of_parents;
    // references to the children, adjacent and parents of
    // the triangle
    struct TRIANGLE **children, *adjacent[3], *parents[2];
    // triangle index in vector
    int triangle_index;
    // circumcenter coordinates of the circunference defined
    // from the 3 points of the triangle
    double xc, yc;
    // circunference radius
    double r;
    // flags
    bool processed, visited;
};
```

ANNEX B - GLOBAL VARIABLES

```
// points vector
vector <struct POINT> points;
// 3 points of enclosing triangle
struct POINT p_0, p_1, p_2;
// root node of triangles tree
struct TRIANGLE *triangles_tree;
// flags to indicate if point is on an edge
bool same_slope_ij, same_slope_jk, same_slope_ki;
// Delaunay triangles vector
vector <struct TRIANGLE *> delaunay_triangles;
// error value for vertices
const double epsVertex = 1.e-5;
// error value for edges
const double epsEdge = 1.e-6;
// constant undefined value
const double UNDEF = -9999.;
// base coordinate X, minimum value of x
const double BASE_X = 0.;
// base coordinate Y, minimum value of y
const double BASE_Y = 0.;
```

ANNEX C - ENCLOSING TRIANGLE

```
// define the coordinates of the points p0,p1,p2 of the
// enclosing triangle of a set of points p in the plane
//
// the coordinates are calculated relative to an
// enclosing rectangle
//
// with the maximum point (max_x+INC,max_y+INC) of the
// rectangle defined, the points (0,K) and (K,0) of the
// triangle are calculated from the intersection of the
// line with slope = -1 (-45o) that passes through the
// maximum point and intersects the x and y axes.
// the last point is defined with coordinates (-K,-K)
void define_enclosing_triangle(vector <struct POINT> &p)
{
    int i;
    double max_x, max_y, K, INC;
    struct TRIANGLE t;

    // define the increment in coordinates
    INC = 10.;

    // define maximum coordinates in x and y
    max_x = -1.;
    max_y = -1.;
    for(i = 0;i < p.size();i++)
    {
        if(p[i].x > max_x)
            max_x = p[i].x;
        if(p[i].y > max_y)
            max_y = p[i].y;
    }

    // define maximum coordinates of the enclosing
    // rectangle incremented by INC shifting up and right
    max_x += INC;
    max_y += INC;

    // define coordinates of points p0,p1,p2 with
    // K = max_x + max_y given by the equation
    // (K - max_y)/(0 - max_x) = -1 of the line that passes
    // through the points (0,K) and (max_x,max_y) with
    // slope = -1
    K = max_x + max_y;
    // multiply K by a constant
    K *= INC;

    // coordinates of points p0,p1,p2
    p_0.x = -K;
    p_0.y = -K;
    p_1.x = 0.;
    p_1.y = K;
    p_2.x = K;
    p_2.y = 0.;

    // define references of the enclosing triangle points
    p_0.address = &p_0;
    p_1.address = &p_1;
    p_2.address = &p_2;

    // define enclosing triangle p0,p1,p2
    t.p_i = p_0;
    t.p_j = p_1;
    t.p_k = p_2;
    t.number_of_children = 0;
    t.children = NULL;
    t.number_of_parents = 0;
    for(i = 0;i < 2;i++)
        t.parents[i] = NULL;
    for(i = 0;i < 3;i++)
        t.adjacent[i] = NULL;
    t.triangle_index = -1;
    t.processed = false;
    t.visited = false;

    // allocate memory to reference a tree of triangles
```

```
    triangles_tree = (struct TRIANGLE *) malloc(sizeof(struct TRIANGLE));
    // define triangle p0,p1,p2 as child of first tree node
    triangles_tree->children = (struct TRIANGLE **) malloc(sizeof(struct TRIANGLE *));
    triangles_tree->children[0] = (struct TRIANGLE *) malloc(sizeof(struct TRIANGLE));
    *(triangles_tree->children[0]) = t;
    triangles_tree->number_of_children = 1;
    triangles_tree->processed = false;
    triangles_tree->visited = false;

    return;
}
```

ANNEX D - DELAUNAY TRIANGULATION

```
// returns flag indicating if triangle has p0, p1 or p2 as vertex
bool valid_delaunay_triangle(struct TRIANGLE *t)
{
    if(t == NULL)
        return(false);

    if((points_are_equal(t->p_i, p_0) == true) ||
       (points_are_equal(t->p_i, p_1) == true) ||
       (points_are_equal(t->p_i, p_2) == true) ||
       (points_are_equal(t->p_j, p_0) == true) ||
       (points_are_equal(t->p_j, p_1) == true) ||
       (points_are_equal(t->p_j, p_2) == true) ||
       (points_are_equal(t->p_k, p_0) == true) ||
       (points_are_equal(t->p_k, p_1) == true) ||
       (points_are_equal(t->p_k, p_2) == true))
        return(false);

    return(true);
}


// perform a depth-first search to select the Delaunay triangles from the leaf nodes of the
    triangles tree
void define_delaunay_triangles_from_tree(struct TRIANGLE *t, vector <struct TRIANGLE *> &
    triangles)
{
    int i;

    // if triangle is not visited
    if(t->visited == false)
    {
        // sets triangle as visited
        t->visited = true;

        // if triangle has children is not leaf, search continues
        if(t->number_of_children > 0)
        {
            for(i = 0;i < t->number_of_children;i++)
                define_delaunay_triangles_from_tree(t->children[i], triangles);
        }
        // if triangle has no children, then it is a Delaunay triangle
        else
        {
            // verifies if triangle does not include points p0,p1,p2 from enclosing triangle
            if(valid_delaunay_triangle(t) == true)
            {
                // inserts triangle in the Delaunay triangles vector
                triangles.push_back(t);
            }
        }
    }

    return;
}

// an edge p1,p2 is legalized by calculating the center and radius of the circunference from
    points p,p1,p2 and by verifing if the point of the adjacent triangle opposite to the edge is
     inside the circunference, so the edge is illegal and it is rotated creating 2 new triangles
void legalize_edge(struct POINT p, struct POINT p1, struct POINT p2, struct TRIANGLE *t_o,
    struct TRIANGLE *t_p)
{
    double xc, yc, r, dist;
    struct POINT opposite_point, center, useless_point_seq[3];
    int i;

    // cases where some points should be excluded from the set
    //
    // if point p is equal to edge point
    if(points_are_equal(p, p1) == true)
    {
        cout << "point␣p␣equals␣to␣p1" << endl;
        cout << "p.index␣=␣" << p.point_index << endl;
        cout << "p1.index␣=␣" << p1.point_index << endl;
        cout << setprecision(15) << "p.x␣=␣" << p.x + BASE_X << "␣|␣p.y␣=␣" << p.y + BASE_Y << "
            ␣|␣p.z␣=␣" << p.z << endl;
```

```
    cout << setprecision(15) << "p1.x = " << p1.x + BASE_X << " | p1.y = " << p1.y + BASE_Y
        << " | p1.z = " << p1.z << endl;
    system("pause");
    exit(0);
    return;
}
// if point p is equal to edge point
if(points_are_equal(p, p2) == true)
{
    cout << "point p equals to p2" << endl;
    cout << "p.index = " << p.point_index << endl;
    cout << "p2.index = " << p2.point_index << endl;
    cout << setprecision(15) << "p.x = " << p.x + BASE_X << " | p.y = " << p.y + BASE_Y << "
        | p.z = " << p.z << endl;
    cout << setprecision(15) << "p2.x = " << p2.x + BASE_X << " | p2.y = " << p2.y + BASE_Y
        << " | p2.z = " << p2.z << endl;
    system("pause");
    exit(0);
    return;
}
// if points p1 and p2 are equal
if(points_are_equal(p1, p2) == true)
{
    cout << "point p1 equals to p2" << endl;
    cout << "p1.index = " << p1.point_index << endl;
    cout << "p2.index = " << p2.point_index << endl;
    cout << setprecision(15) << "p1.x = " << p1.x + BASE_X << " | p1.y = " << p1.y + BASE_Y
        << " | p1.z = " << p1.z << endl;
    cout << setprecision(15) << "p2.x = " << p2.x + BASE_X << " | p2.y = " << p2.y + BASE_Y
        << " | p2.z = " << p2.z << endl;
    system("pause");
    exit(0);
    return;
}
// if p,p1,p2 are colinear in x the circunference does not exist
if(abs(p.x - p1.x) < epsVertex && abs(p1.x - p2.x) < epsVertex)
{
    cout << "3 colinear points, vertical line" << endl;
    cout << setprecision(15) << "p.x =  " << p.point_index << " " << p.x + BASE_X <<  " | p.
        y = " << p.y + BASE_Y << endl;
    cout << setprecision(15) << "p1.x = " << p1.point_index << " " << p1.x + BASE_X << " |
        p1.y = " << p1.y + BASE_Y << endl;
    cout << setprecision(15) << "p2.x = " << p2.point_index << " " << p2.x + BASE_X << " |
        p2.y = " << p2.y + BASE_Y << endl;
    system("pause");
    exit(0);
}
// if p,p1,p2 are colinear in y the circunference does not exist
if(abs(p.y - p1.y) < epsVertex && abs(p1.y - p2.y) < epsVertex)
{
    cout << "3 colinear points, horizontal line" << endl;
    cout << setprecision(15) << "p.x =  " << p.point_index << " " << p.x + BASE_X <<  " | p.
        y = " << p.y + BASE_Y << endl;
    cout << setprecision(15) << "p1.x = " << p1.point_index << " " << p1.x + BASE_X << " |
        p1.y = " << p1.y + BASE_Y << endl;
    cout << setprecision(15) << "p2.x = " << p2.point_index << " " << p2.x + BASE_X << " |
        p2.y = " << p2.y + BASE_Y << endl;
    system("pause");
    exit(0);
}
// if p,p1,p2 are colinear with same slope the circunference does not exist
if(same_slope(p, p1, p2) == true)
{
    cout << "3 colinear points, same slope" << endl;
    cout << setprecision(15) << "p.x =  " << p.x + BASE_X <<  " | p.y = " << p.y + BASE_Y <<
         endl;
    cout << setprecision(15) << "p1.x = " << p1.x + BASE_X << " | p1.y = " << p1.y + BASE_Y
        << endl;
    cout << setprecision(15) << "p2.x = " << p2.x + BASE_X << " | p2.y = " << p2.y + BASE_Y
        << endl;
    system("pause");
    exit(0);
}

// define point opposite to the edge p1,p2 from adjacent triangle
opposite_point = opposite_point_by_edge(p1, p2, t_o);
// if point is not defined
```

```
if(point_not_undef(opposite_point) == false)
    return;

// calculates center and radius of circunference passing through points p,p1,p2
circle_by_3_points(p, p1, p2, &xc, &yc, &r);

// verifies if point opposite to the edge p1,p2 is inside the circunference
//
// calculates distance from circunference center to the opposite point
center.x = xc;
center.y = yc;
dist = distance_between(center, opposite_point);

// verifies if distance is less than radius, so the edge is not valid
if(dist < r)
{
        // divide current triangles adding new triangles p,p1,opposite and p,p2,opposite as
             children
        //
        // triangle with point p
        t_p->children = (struct TRIANGLE **) malloc(2*sizeof(struct TRIANGLE *));
        t_p->number_of_children = 2;
        for(i = 0;i < t_p->number_of_children;i++)
            t_p->children[i] = (struct TRIANGLE *) malloc(sizeof(struct TRIANGLE));

        // triangle p,p1,opposite
        t_p->children[0]->p_i = p;
        t_p->children[0]->p_j = p1;
        t_p->children[0]->p_k = opposite_point;
        t_p->children[0]->children = NULL;
        t_p->children[0]->number_of_children = 0;
        t_p->children[0]->parents[0] = t_p;
        t_p->children[0]->parents[1] = t_o;
        t_p->children[0]->number_of_parents = 2;
        t_p->children[0]->processed = false;
        t_p->children[0]->visited = false;
        // triangle p,p2,opposite
        t_p->children[1]->p_i = p;
        t_p->children[1]->p_j = p2;
        t_p->children[1]->p_k = opposite_point;
        t_p->children[1]->children = NULL;
        t_p->children[1]->number_of_children = 0;
        t_p->children[1]->parents[0] = t_p;
        t_p->children[1]->parents[1] = t_o;
        t_p->children[1]->number_of_parents = 2;
        t_p->children[1]->processed = false;
        t_p->children[1]->visited = false;

        // define adjacency between new triangles
        //
        // triangle p,p1,opposite
        // adjacent by parent p1,p2,p
        t_p->children[0]->adjacent[0] = adjacent_triangle_by_parent(t_p, t_p->children[0],
             p1, p, useless_point_seq);
        // adjacent by parent p1,p2,opposite
        t_p->children[0]->adjacent[1] = adjacent_triangle_by_parent(t_o, t_p->children[0],
             p1, opposite_point, useless_point_seq);
        // adjacent p,p2,opposite
        t_p->children[0]->adjacent[2] = t_p->children[1];
        //
        // triangle p,p2,opposite
        // adjacente pelo triângulo pai p1,p2,p
        t_p->children[1]->adjacent[0] = adjacent_triangle_by_parent(t_p, t_p->children[1],
             p2, p, useless_point_seq);
        // adjacent by parent p1,p2,opposite
        t_p->children[1]->adjacent[1] = adjacent_triangle_by_parent(t_o, t_p->children[1],
             p2, opposite_point, useless_point_seq);
        // adjacent p,p1,opposite
        t_p->children[1]->adjacent[2] = t_p->children[0];

        // triangle with opposite point, children are the same of triangle with point p
        t_o->children = (struct TRIANGLE **) malloc(2*sizeof(struct TRIANGLE *));
        t_o->number_of_children = 2;

        // triangle p,p1,opposite
        t_o->children[0] = t_p->children[0];
        // triangle p,p2,opposite
```

```
            t_o->children[1] = t_p->children[1];

            // recursively legalize edges p1,opposite e opposite,p2
            //
            // adjacent triangles of new triangles
            // adjacent by edge p1,opposite
            legalize_edge(p, p1, opposite_point, t_o->children[0]->adjacent[1], t_o->children
                [0]);
            // adjacent by edge p2,opposite
            legalize_edge(p, opposite_point, p2, t_o->children[1]->adjacent[1], t_o->children
                [1]);
    }

    return;
}

// calculates the Delaunay triangulation from a set of points with the incremental algorithm
//
// the points are inserted one at a time and the triangulation is locally modified with the
    insertion of each point
vector <struct TRIANGLE *> delaunay_triangulation(vector <struct POINT> &p)
{
    vector <struct TRIANGLE *> triangles;
    queue <struct TRIANGLE *> tq;
    struct TRIANGLE *t_node, *t_adjacent;
    struct POINT point_seq[3], useless_point_seq[3];
    int point_in_triangle_result, p_index, t_index;
    int i;

    // inserts a point and verifies the triangle that contains the point
    for(p_index = 0;p_index < p.size();p_index++)
    {
        // search the triangle that contains the point traversing the tree
        //
        // sets the enclosing triangle as not processed and inserts it into the queue
        triangles_tree->children[0]->processed = false;
        tq.push(triangles_tree->children[0]);

        // while the queue is not empty, triangles are still verified
        while(tq.empty() == false)
        {
            // remove triangle from queue
            t_node = tq.front();
            tq.pop();

            // verifies if triangle is already processed, which can occur when the point is
                located exactly on an edge
            //
            // if triangle is already processed, continue, otherwise sets it as processed
            if(t_node->processed == true)
                continue;
            else
                t_node->processed = true;

            // get location of point in the triangle
            point_in_triangle_result = point_in_triangle(p[p_index], *t_node);

            // if the triangle has children, add the children into the queue if the point is on
                the triangle
            if(t_node->number_of_children > 0)
            {
                // if point is inside or on an edge of the triangle
                if(point_in_triangle_result > 0)
                {
                    // no need to verify other triangles in the queue, empty the queue
                    tq = queue<struct TRIANGLE *>();

                    // add children in the queue
                    for(t_index = 0;t_index < t_node->number_of_children;t_index++)
                    {
                        // sets triangle as not processed
                        t_node->children[t_index]->processed = false;
                        tq.push(t_node->children[t_index]);
                    }
                }
            }
            // if triangle is leaf node, divide triangle according to location of point in
```

```
    triangle
else
{
    // if point is inside triangle
    if(point_in_triangle_result == 2)
    {
        // add edges from the point to the vertices of the triangle dividing it into
            3 new triangles
        t_node->children = (struct TRIANGLE **) malloc(3*sizeof(struct TRIANGLE *));
        t_node->number_of_children = 3;
        for(i = 0;i < t_node->number_of_children;i++)
            t_node->children[i] = (struct TRIANGLE *) malloc(sizeof(struct TRIANGLE)
                );

        // triangle i,j,p
        t_node->children[0]->p_i = t_node->p_i;
        t_node->children[0]->p_j = t_node->p_j;
        t_node->children[0]->p_k = p[p_index];
        t_node->children[0]->children = NULL;
        t_node->children[0]->number_of_children = 0;
        t_node->children[0]->parents[0] = t_node;
        t_node->children[0]->number_of_parents = 1;
        t_node->children[0]->processed = false;
        t_node->children[0]->visited = false;
        // triangle j,k,p
        t_node->children[1]->p_i = t_node->p_j;
        t_node->children[1]->p_j = t_node->p_k;
        t_node->children[1]->p_k = p[p_index];
        t_node->children[1]->children = NULL;
        t_node->children[1]->number_of_children = 0;
        t_node->children[1]->parents[0] = t_node;
        t_node->children[1]->number_of_parents = 1;
        t_node->children[1]->processed = false;
        t_node->children[1]->visited = false;
        // triangle i,k,p
        t_node->children[2]->p_i = t_node->p_i;
        t_node->children[2]->p_j = t_node->p_k;
        t_node->children[2]->p_k = p[p_index];
        t_node->children[2]->children = NULL;
        t_node->children[2]->number_of_children = 0;
        t_node->children[2]->parents[0] = t_node;
        t_node->children[2]->number_of_parents = 1;
        t_node->children[2]->processed = false;
        t_node->children[2]->visited = false;

        // define adjacency between new triangles
        //
        // triangle i,j,p:
        // adjacent by parent
        t_node->children[0]->adjacent[0] = adjacent_triangle_by_parent(t_node,
            t_node->children[0], t_node->p_i, t_node->p_j, useless_point_seq);
        // adjacent j,k,p
        t_node->children[0]->adjacent[1] = t_node->children[1];
        // adjacent i,k,p
        t_node->children[0]->adjacent[2] = t_node->children[2];
        //
        // triangle j,k,p:
        // adjacent by parent
        t_node->children[1]->adjacent[0] = adjacent_triangle_by_parent(t_node,
            t_node->children[1], t_node->p_j, t_node->p_k, useless_point_seq);
        // adjacent i,k,p
        t_node->children[1]->adjacent[1] = t_node->children[2];
        // adjacent i,j,p
        t_node->children[1]->adjacent[2] = t_node->children[0];
        //
        // triangle i,k,p:
        // adjacent by parent
        t_node->children[2]->adjacent[0] = adjacent_triangle_by_parent(t_node,
            t_node->children[2], t_node->p_i, t_node->p_k, useless_point_seq);
        // adjacent j,k,p
        t_node->children[2]->adjacent[1] = t_node->children[1];
        // adjacent i,j,p
        t_node->children[2]->adjacent[2] = t_node->children[0];

        // legalize the edges of the divided triangle
        legalize_edge(p[p_index], t_node->p_i, t_node->p_j, t_node->children[0]->
            adjacent[0], t_node->children[0]);
```

```
        legalize_edge(p[p_index], t_node->p_j, t_node->p_k, t_node->children[1]->
            adjacent[0], t_node->children[1]);
        legalize_edge(p[p_index], t_node->p_k, t_node->p_i, t_node->children[2]->
            adjacent[0], t_node->children[2]);

        // no need to verify other triangles in the queue, empty the queue
        tq = queue<struct TRIANGLE *>();
    }
    // if the point is on an edge
    else
    if(point_in_triangle_result == 1)
    {
        // add an edge from the point to the point of the triangle opposite to the
            edge dividing it into 2 new triangles
        t_node->children = (struct TRIANGLE **) malloc(2*sizeof(struct TRIANGLE *));
        t_node->number_of_children = 2;
        for(i = 0;i < t_node->number_of_children;i++)
            t_node->children[i] = (struct TRIANGLE *) malloc(sizeof(struct TRIANGLE)
                );

        // verifies on what edge the point is located and defines the new triangles
            with the opposite point
        //
        // edge i,j
        // adjacent triangle: point l <-> point k
        if(same_slope_ij == true)
        {
            // triangle i,k,p
            t_node->children[0]->p_i = t_node->p_i;
            t_node->children[0]->p_j = t_node->p_k;
            t_node->children[0]->p_k = p[p_index];
            t_node->children[0]->children = NULL;
            t_node->children[0]->number_of_children = 0;
            t_node->children[0]->parents[0] = t_node;
            t_node->children[0]->number_of_parents = 1;
            t_node->children[0]->processed = false;
            t_node->children[0]->visited = false;
            // triangle j,k,p
            t_node->children[1]->p_i = t_node->p_j;
            t_node->children[1]->p_j = t_node->p_k;
            t_node->children[1]->p_k = p[p_index];
            t_node->children[1]->children = NULL;
            t_node->children[1]->number_of_children = 0;
            t_node->children[1]->parents[0] = t_node;
            t_node->children[1]->number_of_parents = 1;
            t_node->children[1]->processed = false;
            t_node->children[1]->visited = false;

            // define 2 new triangles from the adjacent triangle by the edge
            t_adjacent = adjacent_triangle_by_parent(t_node, t_node, t_node->p_i,
                t_node->p_j, point_seq);

            if(t_adjacent != NULL)
            {
                t_adjacent->children = (struct TRIANGLE **) malloc(2*sizeof(struct
                    TRIANGLE *));
                t_adjacent->number_of_children = 2;
                for(i = 0;i < t_adjacent->number_of_children;i++)
                    t_adjacent->children[i] = (struct TRIANGLE *) malloc(sizeof(
                        struct TRIANGLE));

                // sets adjacent triangle as processed
                t_adjacent->processed = true;

                // triangle i,l,p
                t_adjacent->children[0]->p_i = point_seq[1];
                t_adjacent->children[0]->p_j = point_seq[0];
                t_adjacent->children[0]->p_k = p[p_index];
                t_adjacent->children[0]->children = NULL;
                t_adjacent->children[0]->number_of_children = 0;
                t_adjacent->children[0]->parents[0] = t_adjacent;
                t_adjacent->children[0]->number_of_parents = 1;
                t_adjacent->children[0]->processed = false;
                t_adjacent->children[0]->visited = false;
                // triangle j,l,p
                t_adjacent->children[1]->p_i = point_seq[2];
                t_adjacent->children[1]->p_j = point_seq[0];
```

```
            t_adjacent->children[1]->p_k = p[p_index];
            t_adjacent->children[1]->children = NULL;
            t_adjacent->children[1]->number_of_children = 0;
            t_adjacent->children[1]->parents[0] = t_adjacent;
            t_adjacent->children[1]->number_of_parents = 1;
            t_adjacent->children[1]->processed = false;
            t_adjacent->children[1]->visited = false;
        }

        // define adjacency between new triangles
        //
        // triangle i,k,p:
        // adjacent by parent
        t_node->children[0]->adjacent[0] = adjacent_triangle_by_parent(t_node,
             t_node->children[0], t_node->p_i, t_node->p_k, useless_point_seq);
        // adjacent j,k,p
        t_node->children[0]->adjacent[1] = t_node->children[1];
        // adjacent i,l,p
        if(t_adjacent != NULL)
            t_node->children[0]->adjacent[2] = t_adjacent->children[0];

        // triangle j,k,p:
        // adjacent by parent
        t_node->children[1]->adjacent[0] = adjacent_triangle_by_parent(t_node,
             t_node->children[1], t_node->p_j, t_node->p_k, useless_point_seq);
        // adjacent i,k,p
        t_node->children[1]->adjacent[1] = t_node->children[0];
        // adjacent j,l,p
        if(t_adjacent != NULL)
            t_node->children[1]->adjacent[2] = t_adjacent->children[1];

        if(t_adjacent != NULL)
        {
            // triangle i,l,p:
            // adjacente pelo triângulo pai
            t_adjacent->children[0]->adjacent[0] = adjacent_triangle_by_parent(
                 t_adjacent, t_adjacent->children[0], point_seq[1], point_seq[0],
                  useless_point_seq);
            // adjacent j,l,p
            t_adjacent->children[0]->adjacent[1] = t_adjacent->children[1];
            // adjacent i,k,p
            t_adjacent->children[0]->adjacent[2] = t_node->children[0];
            //
            // triangle j,l,p:
            // adjacente pelo triângulo pai
            t_adjacent->children[1]->adjacent[0] = adjacent_triangle_by_parent(
                 t_adjacent, t_adjacent->children[1], point_seq[2], point_seq[0],
                  useless_point_seq);
            // adjacent i,l,p
            t_adjacent->children[1]->adjacent[1] = t_adjacent->children[0];
            // adjacent j,k,p
            t_adjacent->children[1]->adjacent[2] = t_node->children[1];
        }

        // legalize the edges of the divided triangle
        legalize_edge(p[p_index], t_node->p_i, point_seq[0], t_adjacent->
             children[0]->adjacent[0], t_adjacent->children[0]);
        legalize_edge(p[p_index], point_seq[0], t_node->p_j, t_adjacent->
             children[1]->adjacent[0], t_adjacent->children[1]);
        legalize_edge(p[p_index], t_node->p_j, t_node->p_k, t_node->children
             [1]->adjacent[0], t_node->children[1]);
        legalize_edge(p[p_index], t_node->p_k, t_node->p_i, t_node->children
             [0]->adjacent[0], t_node->children[0]);
    }
    else
    // edge j,k
    // adjacent triangle: point l <-> point i
    if(same_slope_jk == true)
    {
        // triangle j,i,p
        t_node->children[0]->p_i = t_node->p_j;
        t_node->children[0]->p_j = t_node->p_i;
        t_node->children[0]->p_k = p[p_index];
        t_node->children[0]->children = NULL;
        t_node->children[0]->number_of_children = 0;
        t_node->children[0]->parents[0] = t_node;
        t_node->children[0]->number_of_parents = 1;
```

```
t_node->children[0]->processed = false;
t_node->children[0]->visited = false;
// triangle k,i,p
t_node->children[1]->p_i = t_node->p_k;
t_node->children[1]->p_j = t_node->p_i;
t_node->children[1]->p_k = p[p_index];
t_node->children[1]->children = NULL;
t_node->children[1]->number_of_children = 0;
t_node->children[1]->parents[0] = t_node;
t_node->children[1]->number_of_parents = 1;
t_node->children[1]->processed = false;
t_node->children[1]->visited = false;

// define 2 new triangles from the adjacent triangle by the edge
t_adjacent = adjacent_triangle_by_parent(t_node, t_node, t_node->p_j,
     t_node->p_k, point_seq);

if(t_adjacent != NULL)
{
    t_adjacent->children = (struct TRIANGLE **) malloc(2*sizeof(struct
         TRIANGLE *));
    t_adjacent->number_of_children = 2;
    for(i = 0;i < t_adjacent->number_of_children;i++)
        t_adjacent->children[i] = (struct TRIANGLE *) malloc(sizeof(
             struct TRIANGLE));

    // sets adjacent triangle as processed
    t_adjacent->processed = true;

    // triangle j,l,p
    t_adjacent->children[0]->p_i = point_seq[1];
    t_adjacent->children[0]->p_j = point_seq[0];
    t_adjacent->children[0]->p_k = p[p_index];
    t_adjacent->children[0]->children = NULL;
    t_adjacent->children[0]->number_of_children = 0;
    t_adjacent->children[0]->parents[0] = t_adjacent;
    t_adjacent->children[0]->number_of_parents = 1;
    t_adjacent->children[0]->processed = false;
    t_adjacent->children[0]->visited = false;
    // triangle k,l,p
    t_adjacent->children[1]->p_i = point_seq[2];
    t_adjacent->children[1]->p_j = point_seq[0];
    t_adjacent->children[1]->p_k = p[p_index];
    t_adjacent->children[1]->children = NULL;
    t_adjacent->children[1]->number_of_children = 0;
    t_adjacent->children[1]->parents[0] = t_adjacent;
    t_adjacent->children[1]->number_of_parents = 1;
    t_adjacent->children[1]->processed = false;
    t_adjacent->children[1]->visited = false;
}

// define adjacency between new triangles
//
// triangle j,i,p:
// adjacent by parent
t_node->children[0]->adjacent[0] = adjacent_triangle_by_parent(t_node,
     t_node->children[0], t_node->p_j, t_node->p_i, useless_point_seq);
// adjacent k,i,p
t_node->children[0]->adjacent[1] = t_node->children[1];
// adjacent j,l,p
if(t_adjacent != NULL)
    t_node->children[0]->adjacent[2] = t_adjacent->children[0];

// triângulo k,i,p:
// adjacent by parent
t_node->children[1]->adjacent[0] = adjacent_triangle_by_parent(t_node,
     t_node->children[1], t_node->p_k, t_node->p_i, useless_point_seq);
// adjacent j,i,p
t_node->children[1]->adjacent[1] = t_node->children[0];
// adjacent k,l,p
if(t_adjacent != NULL)
    t_node->children[1]->adjacent[2] = t_adjacent->children[1];

if(t_adjacent != NULL)
{
    // triangle j,l,p:
    // adjacent by parent
```

```
            t_adjacent->children[0]->adjacent[0] = adjacent_triangle_by_parent(
                t_adjacent, t_adjacent->children[0], point_seq[1], point_seq[0],
                 useless_point_seq);
            // adjacent k,l,p
            t_adjacent->children[0]->adjacent[1] = t_adjacent->children[1];
            // adjacent j,i,p
            t_adjacent->children[0]->adjacent[2] = t_node->children[0];
            //
            // triangle k,l,p:
            // adjacente pelo triângulo pai
            t_adjacent->children[1]->adjacent[0] = adjacent_triangle_by_parent(
                t_adjacent, t_adjacent->children[1], point_seq[2], point_seq[0],
                 useless_point_seq);
            // adjacent j,l,p
            t_adjacent->children[1]->adjacent[1] = t_adjacent->children[0];
            // adjacent k,i,p
            t_adjacent->children[1]->adjacent[2] = t_node->children[1];
        }

        // legalize the edges of the divided triangle
        legalize_edge(p[p_index], t_node->p_j, point_seq[0], t_adjacent->
            children[0]->adjacent[0], t_adjacent->children[0]);
        legalize_edge(p[p_index], point_seq[0], t_node->p_k, t_adjacent->
            children[1]->adjacent[0], t_adjacent->children[1]);
        legalize_edge(p[p_index], t_node->p_k, t_node->p_i, t_node->children
            [1]->adjacent[0], t_node->children[1]);
        legalize_edge(p[p_index], t_node->p_i, t_node->p_j, t_node->children
            [0]->adjacent[0], t_node->children[0]);
    }
    else
    // edge k,i
    // adjacent triangle: point l <-> point j
    if(same_slope_ki == true)
    {
        // triangle k,j,p
        t_node->children[0]->p_i = t_node->p_k;
        t_node->children[0]->p_j = t_node->p_j;
        t_node->children[0]->p_k = p[p_index];
        t_node->children[0]->children = NULL;
        t_node->children[0]->number_of_children = 0;
        t_node->children[0]->parents[0] = t_node;
        t_node->children[0]->number_of_parents = 1;
        t_node->children[0]->processed = false;
        t_node->children[0]->visited = false;
        // triangle i,j,p
        t_node->children[1]->p_i = t_node->p_i;
        t_node->children[1]->p_j = t_node->p_j;
        t_node->children[1]->p_k = p[p_index];
        t_node->children[1]->children = NULL;
        t_node->children[1]->number_of_children = 0;
        t_node->children[1]->parents[0] = t_node;
        t_node->children[1]->number_of_parents = 1;
        t_node->children[1]->processed = false;
        t_node->children[1]->visited = false;

        // define 2 new triangles from the adjacent triangle by the edge
        t_adjacent = adjacent_triangle_by_parent(t_node, t_node, t_node->p_k,
            t_node->p_i, point_seq);

        if(t_adjacent != NULL)
        {
            t_adjacent->children = (struct TRIANGLE **) malloc(2*sizeof(struct
                TRIANGLE *));
            t_adjacent->number_of_children = 2;
            for(i = 0;i < t_adjacent->number_of_children;i++)
                t_adjacent->children[i] = (struct TRIANGLE *) malloc(sizeof(
                    struct TRIANGLE));

            // sets adjacent triangle as processed
            t_adjacent->processed = true;

            // triangle k,l,p
            t_adjacent->children[0]->p_i = point_seq[1];
            t_adjacent->children[0]->p_j = point_seq[0];
            t_adjacent->children[0]->p_k = p[p_index];
            t_adjacent->children[0]->children = NULL;
            t_adjacent->children[0]->number_of_children = 0;
```

```
                    t_adjacent->children[0]->parents[0] = t_adjacent;
                    t_adjacent->children[0]->number_of_parents = 1;
                    t_adjacent->children[0]->processed = false;
                    t_adjacent->children[0]->visited = false;
                    // triangle i,l,p
                    t_adjacent->children[1]->p_i = point_seq[2];
                    t_adjacent->children[1]->p_j = point_seq[0];
                    t_adjacent->children[1]->p_k = p[p_index];
                    t_adjacent->children[1]->children = NULL;
                    t_adjacent->children[1]->number_of_children = 0;
                    t_adjacent->children[1]->parents[0] = t_adjacent;
                    t_adjacent->children[1]->number_of_parents = 1;
                    t_adjacent->children[1]->processed = false;
                    t_adjacent->children[1]->visited = false;
                }

                // define adjacency between new triangles
                //
                // triangle k,j,p:
                // adjacent by parent
                t_node->children[0]->adjacent[0] = adjacent_triangle_by_parent(t_node,
                     t_node->children[0], t_node->p_k, t_node->p_j, useless_point_seq);
                // adjacent i,j,p
                t_node->children[0]->adjacent[1] = t_node->children[1];
                // adjacent k,l,p
                if(t_adjacent != NULL)
                    t_node->children[0]->adjacent[2] = t_adjacent->children[0];

                // triangle i,j,p:
                // adjacent by parent
                t_node->children[1]->adjacent[0] = adjacent_triangle_by_parent(t_node,
                     t_node->children[1], t_node->p_i, t_node->p_j, useless_point_seq);
                // adjacent k,j,p
                t_node->children[1]->adjacent[1] = t_node->children[0];
                // adjacent i,l,p
                if(t_adjacent != NULL)
                    t_node->children[1]->adjacent[2] = t_adjacent->children[1];

                if(t_adjacent != NULL)
                {
                    // triangle k,l,p:
                    // adjacent by parent
                    t_adjacent->children[0]->adjacent[0] = adjacent_triangle_by_parent(
                         t_adjacent, t_adjacent->children[0], point_seq[1], point_seq[0],
                          useless_point_seq);
                    // adjacent i,l,p
                    t_adjacent->children[0]->adjacent[1] = t_adjacent->children[1];
                    // adjacent k,j,p
                    t_adjacent->children[0]->adjacent[2] = t_node->children[0];
                    //
                    // triangle i,l,p:
                    // adjacent by parent
                    t_adjacent->children[1]->adjacent[0] = adjacent_triangle_by_parent(
                         t_adjacent, t_adjacent->children[1], point_seq[2], point_seq[0],
                          useless_point_seq);
                    // adjacent k,l,p
                    t_adjacent->children[1]->adjacent[1] = t_adjacent->children[0];
                    // adjacent i,j,p
                    t_adjacent->children[1]->adjacent[2] = t_node->children[1];
                }

                // legalize the edges of the divided triangle
                legalize_edge(p[p_index], t_node->p_k, point_seq[0], t_adjacent->
                     children[0]->adjacent[0], t_adjacent->children[0]);
                legalize_edge(p[p_index], point_seq[0], t_node->p_i, t_adjacent->
                     children[1]->adjacent[0], t_adjacent->children[1]);
                legalize_edge(p[p_index], t_node->p_i, t_node->p_j, t_node->children
                     [1]->adjacent[0], t_node->children[1]);
                legalize_edge(p[p_index], t_node->p_j, t_node->p_k, t_node->children
                     [0]->adjacent[0], t_node->children[0]);
            }

            // no need to verify other triangles in the queue, empty the queue
            tq = queue<struct TRIANGLE *>();
        }
    }
}
```

```
    }

    // perform a DFS on the tree to search leaf nodes which are Delaunay triangles
    define_delaunay_triangles_from_tree(triangles_tree, triangles);

    return(triangles);
}
```

ANNEX E - GEOMETRIC FUNCTIONS

```
// calculates euclidian distance between 2 points in the plane
double distance_between(struct POINT p, struct POINT q)
{
    double dist, dx, dy;

    dx = p.x - q.x;
    dy = p.y - q.y;
    dist = sqrt(dx*dx+dy*dy);

    return(dist);
}

// calculates distance between point p and line Ax+By+C with coefficients A,B,C as |A*x+B*y+C|/
    SQRT(A^2+B^2)
double point_line_distance(double A, double B, double C, struct POINT p)
{
    double distance;

    distance = abs(A*(p.x) + B*(p.y) + C)/sqrt(A*A + B*B);

    return(distance);
}

// verifies if point p is on the line that passes through points p1,p2
//
// uses point-line distance formula by calculating the coefficients A,B,C of the line Ax+By+C=0
    from points p1,p2
bool same_slope(struct POINT p, struct POINT p1, struct POINT p2)
{
    double A, B, C, distance;

    // coefficients of line p1,p2
    A = p1.y - p2.y;
    B = p2.x - p1.x;
    C = (p1.x - p2.x)*p1.y + (p2.y - p1.y)*p1.x;

    // distance between point p and line p1,p2
    distance = point_line_distance(A, B, C, p);

    // if distance is less than an error, then point is on the line
    if(distance < epsEdge)
        return(true);
    else
        return(false);
}

// calculates intersection point between 2 segments p1,p2 and p3,p4
//
// cases of vertical/horizontal lines are treated separately
//
// coordinates of intersection point are the solution of system of 2 equations
struct POINT segment_intersection(struct POINT p1, struct POINT p2, struct POINT p3, struct
    POINT p4)
{
    struct POINT p;
    double xi, yi, m12, m34;

    // parallel lines
    if(same_slope(p1, p3, p4) == true && same_slope(p2, p3, p4) == true)
    {
        p.x = UNDEF;
        p.y = UNDEF;

        return(p);
    }

    // segment p1,p2 is vertical
    if(abs(p1.x - p2.x) < epsVertex)
    {
        // segment p3,p4 is vertical, so both are parallel, there is no intersection point
        if(abs(p3.x - p4.x) < epsVertex)
        {
            p.x = UNDEF;
            p.y = UNDEF;
```

```
        return(p);
    }

    // segment p3,p4 is horizontal
    if(abs(p3.y - p4.y) < epsVertex)
    {
        p.x = p1.x;
        p.y = p3.y;

        if(((-epsVertex + p3.x <= p.x && p.x <= p4.x + epsVertex) || (-epsVertex + p4.x <= p
            .x && p.x <= p3.x + epsVertex)) &&
           ((-epsVertex + p1.y <= p.y && p.y <= p2.y + epsVertex) || (-epsVertex + p2.y <= p
               .y && p.y <= p1.y + epsVertex)))
            return(p);

        p.x = UNDEF;
        p.y = UNDEF;

        return(p);
    }

    // segment p3,p4 is not vertical/horizontal
    xi = p1.x;
    m34 = (p4.y - p3.y)/(p4.x - p3.x);
    yi = p3.y + m34*(xi - p3.x);

    p.x = xi;
    p.y = yi;

    if(((-epsVertex + p3.x <= p.x && p.x <= p4.x + epsVertex) || (-epsVertex + p4.x <= p.x
        && p.x <= p3.x + epsVertex)) &&
       ((-epsVertex + p3.y <= p.y && p.y <= p4.y + epsVertex) || (-epsVertex + p4.y <= p.y
           && p.y <= p3.y + epsVertex)) &&
       ((-epsVertex + p1.y <= p.y && p.y <= p2.y + epsVertex) || (-epsVertex + p2.y <= p.y
           && p.y <= p1.y + epsVertex)))
        return(p);

    p.x = UNDEF;
    p.y = UNDEF;

    return(p);
}

// segment p1,p2 is horizontal
if(abs(p1.y - p2.y) < epsVertex)
{
    // segment p3,p4 is horizontal, so both are parallel, there is no intersection point
    if(abs(p3.y - p4.y) < epsVertex)
    {
        p.x = UNDEF;
        p.y = UNDEF;

        return(p);
    }

    // segment p3,p4 is vertical
    if(abs(p3.x - p4.x) < epsVertex)
    {
        p.x = p3.x;
        p.y = p1.y;

        if(((-epsVertex + p1.x <= p.x && p.x <= p2.x + epsVertex) || (-epsVertex + p2.x <= p
            .x && p.x <= p1.x + epsVertex)) &&
           ((-epsVertex + p3.y <= p.y && p.y <= p4.y + epsVertex) || (-epsVertex + p4.y <= p
               .y && p.y <= p3.y + epsVertex)))
            return(p);

        p.x = UNDEF;
        p.y = UNDEF;

        return(p);
    }

    // segment p3,p4 is not vertical/horizontal
    yi = p1.y;
    m34 = (p4.y - p3.y)/(p4.x - p3.x);
    xi = (yi - p3.y)/m34 + p3.x;
```

```
    p.x = xi;
    p.y = yi;

    if(((-epsVertex + p3.x <= p.x && p.x <= p4.x + epsVertex) || (-epsVertex + p4.x <= p.x
        && p.x <= p3.x + epsVertex)) &&
       ((-epsVertex + p3.y <= p.y && p.y <= p4.y + epsVertex) || (-epsVertex + p4.y <= p.y
           && p.y <= p3.y + epsVertex)) &&
       ((-epsVertex + p1.x <= p.x && p.x <= p2.x + epsVertex) || (-epsVertex + p2.x <= p.x
           && p.x <= p1.x + epsVertex)))
        return(p);

    p.x = UNDEF;
    p.y = UNDEF;

    return(p);
}

// segment p1,p2 is not vertical/horizontal
//
// segment p3,p4 is vertical
if(abs(p3.x - p4.x) < epsVertex)
{
    xi = p3.x;
    m12 = (p2.y - p1.y)/(p2.x - p1.x);
    yi = p1.y + m12*(xi - p1.x);

    p.x = xi;
    p.y = yi;

    if(((-epsVertex + p1.x <= p.x && p.x <= p2.x + epsVertex) || (-epsVertex + p2.x <= p.x
        && p.x <= p1.x + epsVertex)) &&
       ((-epsVertex + p1.y <= p.y && p.y <= p2.y + epsVertex) || (-epsVertex + p2.y <= p.y
           && p.y <= p1.y + epsVertex)) &&
       ((-epsVertex + p3.y <= p.y && p.y <= p4.y + epsVertex) || (-epsVertex + p4.y <= p.y
           && p.y <= p3.y + epsVertex)))
        return(p);

    p.x = UNDEF;
    p.y = UNDEF;

    return(p);
}
// segment p3,p4 is horizontal
if(abs(p3.y - p4.y) < epsVertex)
{
    yi = p3.y;
    m12 = (p2.y - p1.y)/(p2.x - p1.x);
    xi = (yi - p1.y)/m12 + p1.x;

    p.x = xi;
    p.y = yi;

    if(((-epsVertex + p1.x <= p.x && p.x <= p2.x + epsVertex) || (-epsVertex + p2.x <= p.x
        && p.x <= p1.x + epsVertex)) &&
       ((-epsVertex + p1.y <= p.y && p.y <= p2.y + epsVertex) || (-epsVertex + p2.y <= p.y
           && p.y <= p1.y + epsVertex)) &&
       ((-epsVertex + p3.x <= p.x && p.x <= p4.x + epsVertex) || (-epsVertex + p4.x <= p.x
           && p.x <= p3.x + epsVertex)))
        return(p);

    p.x = UNDEF;
    p.y = UNDEF;

    return(p);
}

// both segments are not horizontal/vertical
m12 = (p2.y - p1.y)/(p2.x - p1.x);
m34 = (p4.y - p3.y)/(p4.x - p3.x);
xi = (p3.y - p1.y - m34*p3.x + m12*p1.x)/(m12 - m34);
yi = p1.y + m12*(xi - p1.x);

p.x = xi;
p.y = yi;

if(((-epsVertex + p1.x <= p.x && p.x <= p2.x + epsVertex) || (-epsVertex + p2.x <= p.x && p.
```

```
          x <= p1.x + epsVertex)) &&
        ((-epsVertex + p1.y <= p.y && p.y <= p2.y + epsVertex) || (-epsVertex + p2.y <= p.y && p.
            y <= p1.y + epsVertex)) &&
        ((-epsVertex + p3.x <= p.x && p.x <= p4.x + epsVertex) || (-epsVertex + p4.x <= p.x && p.
            x <= p3.x + epsVertex)) &&
        ((-epsVertex + p3.y <= p.y && p.y <= p4.y + epsVertex) || (-epsVertex + p4.y <= p.y && p.
            y <= p3.y + epsVertex)))
         return(p);

    p.x = UNDEF;
    p.y = UNDEF;

    return(p);
}

// verifies if point is inside triangle, on an edge or outside triangle
//
// considers a horizontal line from the point going to the right and checks how many and what
     kind of intersections with edges occur
//
// return values
// 0 - point is outside triangle
// 1 - point is on an edge
// 2 - point is inside triangle
int point_in_triangle(struct POINT p, struct TRIANGLE t)
{
    struct POINT pmax, intersection_point;
    double max_x;
    int edge_intersections;
    bool up, down;

    // if point is on the edge i,j
    if(same_slope(p, t.p_i, t.p_j) == true)
    {
        same_slope_ij = true;
        same_slope_jk = same_slope_ki = false;

        // if coordinates of the point are in the range of the coordinates of the edge points
        if(((-epsVertex + t.p_i.x <= p.x && p.x <= t.p_j.x + epsVertex) || (-epsVertex + t.p_j.x
              <= p.x && p.x <= t.p_i.x + epsVertex)) &&
           ((-epsVertex + t.p_i.y <= p.y && p.y <= t.p_j.y + epsVertex) || (-epsVertex + t.p_j.y
                <= p.y && p.y <= t.p_i.y + epsVertex)))
            return(1);
        else
            return(0);
    }

    // if point is on the edge j,k
    if(same_slope(p, t.p_j, t.p_k) == true)
    {
        same_slope_jk = true;
        same_slope_ki = same_slope_ij = false;

        // if coordinates of the point are in the range of the coordinates of the edge points
        if(((-epsVertex + t.p_j.x <= p.x && p.x <= t.p_k.x + epsVertex) || (-epsVertex + t.p_k.x
              <= p.x && p.x <= t.p_j.x + epsVertex)) &&
           ((-epsVertex + t.p_j.y <= p.y && p.y <= t.p_k.y + epsVertex) || (-epsVertex + t.p_k.y
                <= p.y && p.y <= t.p_j.y + epsVertex)))
            return(1);
        else
            return(0);
    }

    // if point is on the edge k,i
    if(same_slope(p, t.p_k, t.p_i) == true)
    {
        same_slope_ki = true;
        same_slope_ij = same_slope_jk = false;

        // if coordinates of the point are in the range of the coordinates of the edge points
        if(((-epsVertex + t.p_k.x <= p.x && p.x <= t.p_i.x + epsVertex) || (-epsVertex + t.p_i.x
              <= p.x && p.x <= t.p_k.x + epsVertex)) &&
           ((-epsVertex + t.p_k.y <= p.y && p.y <= t.p_i.y + epsVertex) || (-epsVertex + t.p_i.y
                <= p.y && p.y <= t.p_k.y + epsVertex)))
            return(1);
        else
            return(0);
```

```
}

// defines maximum x coordinate from points i,j,k of the triangle
max_x = -1.;
if(t.p_i.x > max_x)
    max_x = t.p_i.x;
if(t.p_j.x > max_x)
    max_x = t.p_j.x;
if(t.p_k.x > max_x)
    max_x = t.p_k.x;

// coordinates of the outside point pmax
pmax.x = max_x + INC;
pmax.y = p.y;

// verifies if point is inside triangle by the intersection between horizontal line from
    point p with the triangle edges
//
// initialize number of intersections and direction of edge points
edge_intersections = 0;
up = down = false;
// intersection between segments p,pmax and i,j
intersection_point = segment_intersection(p, pmax, t.p_i, t.p_j);
// if intersection exists
if(point_not_undef(intersection_point) == true)
{
    // increment number of intersections
    edge_intersections++;

    // if intersection is a vertex
    if(points_are_equal(t.p_i, intersection_point) == true)
    {
        // if intersected vertex is up or down on the edge
        if(t.p_i.y < t.p_j.y)
            down = true;
        else
            up = true;
    }
    else
    // if intersection is a vertex
    if(points_are_equal(t.p_j, intersection_point) == true)
    {
        // if intersected vertex is up or down on the edge
        if(t.p_j.y < t.p_i.y)
            down = true;
        else
            up = true;
    }
}

// intersection between segments p,pmax and j,k
intersection_point = segment_intersection(p, pmax, t.p_j, t.p_k);
// if intersection exists
if(point_not_undef(intersection_point) == true)
{
    // increment number of intersections
    edge_intersections++;

    // if intersection is a vertex
    if(points_are_equal(t.p_j, intersection_point) == true)
    {
        // if intersected vertex is up or down on the edge
        if(t.p_j.y < t.p_k.y)
            down = true;
        else
            up = true;
    }
    else
    // if intersection is a vertex
    if(points_are_equal(t.p_k, intersection_point) == true)
    {
        // if intersected vertex is up or down on the edge
        if(t.p_k.y < t.p_j.y)
            down = true;
        else
            up = true;
    }
```

```
    }

    // intersection between segments p,pmax and k,i
    intersection_point = segment_intersection(p, pmax, t.p_k, t.p_i);
    // if intersection exists
    if(point_not_undef(intersection_point) == true)
    {
        // increment number of intersections
        edge_intersections++;

        // if intersection is a vertex
        if(points_are_equal(t.p_k, intersection_point) == true)
        {
            // if intersected vertex is up or down on the edge
            if(t.p_k.y < t.p_i.y)
                down = true;
            else
                up = true;
        }
        else
        // if intersection is a vertex
        if(points_are_equal(t.p_i, intersection_point) == true)
        {
            // if intersected vertex is up or down on the edge
            if(t.p_i.y < t.p_k.y)
                down = true;
            else
                up = true;
        }
    }

    // if number of intersections with edges is either 1 or 2 and it happened with a vertex
        forming a right "beak", then point is inside triangle
    if((edge_intersections == 1) || (edge_intersections == 2 && down == true && up == true))
        return(2);

    return(0);
}

// calculates the center and radius of a circunference passing through points p,p1,p2
void circle_by_3_points(struct POINT p, struct POINT p1, struct POINT p2, double *xc, double *yc
    , double *r)
{
    double dx, dy;
    double m1, m2, xm1, ym1, xm2, ym2;

    // verifies if line p1,p2 is vertical and p2,p is horizontal, then circumcenter is defined
        by mean point
    if(abs(p1.x - p2.x) < epsVertex && abs(p2.y - p.y) < epsVertex)
    {
        // calculates circumcenter coordinates
        *xc = (p2.x + p.x)/2;
        *yc = (p1.y + p2.y)/2;
    }
    else
    // verifies if line p2,p is vertical and p,p1 is horizontal, then circumcenter is defined by
        mean point
    if(abs(p2.x - p.x) < epsVertex && abs(p.y - p1.y) < epsVertex)
    {
        // calculates circumcenter coordinates
        *xc = (p.x + p1.x)/2;
        *yc = (p2.y + p.y)/2;
    }
    else
    // verifies if line p,p1 is vertical and p1,p2 is horizontal, then circumcenter is defined
        by mean point
    if(abs(p.x - p1.x) < epsVertex && abs(p1.y - p2.y) < epsVertex)
    {
        // calculates circumcenter coordinates
        *xc = (p1.x + p2.x)/2;
        *yc = (p.y + p1.y)/2;
    }
    else
    // verifies if line p1,p2 is horizontal and p2,p is vertical, then circumcenter is defined
        by mean point
    if(abs(p1.y - p2.y) < epsVertex && abs(p2.x - p.x) < epsVertex)
    {
```

```
    // calculates circumcenter coordinates
    *xc = (p1.x + p2.x)/2;
    *yc = (p2.y + p.y)/2;
}
else
// verifies if line p2,p is horizontal and p,p1 is vertical, then circumcenter is defined by
    mean point
if(abs(p2.y - p.y) < epsVertex && abs(p.x - p1.x) < epsVertex)
{
    // calculates circumcenter coordinates
    *xc = (p2.x + p.x)/2;
    *yc = (p.y + p1.y)/2;
}
else
// verifies if line p,p1 is horizontal and p1,p2 is vertical, then circumcenter is defined
    by mean point
if(abs(p.y - p1.y) < epsVertex && abs(p1.x - p2.x) < epsVertex)
{
    // calculates circumcenter coordinates
    *xc = (p.x + p1.x)/2;
    *yc = (p1.y + p2.y)/2;
}
else
// verifies if line p1,p2 is vertical, then circumcenter is defined by constant line
if(abs(p1.x - p2.x) < epsVertex)
{
    // calculates circumcenter coordinates
    // vertical line, calculates mean y and x by intersection
    *yc = (p1.y + p2.y)/2;
    xm2 = (p2.x + p.x)/2;
    ym2 = (p2.y + p.y)/2;
    m2 = (p.y - p2.y)/(p.x - p2.x);
    *xc = -(*yc - ym2)*m2 + xm2;
}
else
// verifies if line p2,p is vertical, then circumcenter is defined by constant line
if(abs(p2.x - p.x) < epsVertex)
{
    // calculates circumcenter coordinates
    // vertical line, calculates mean y and x by intersection
    *yc = (p2.y + p.y)/2;
    xm1 = (p1.x + p2.x)/2;
    ym1 = (p1.y + p2.y)/2;
    m1 = (p2.y - p1.y)/(p2.x - p1.x);
    *xc = -(*yc - ym1)*m1 + xm1;
}
else
// verifies if line p,p1 is vertical, then circumcenter is defined by constant line
if(abs(p.x - p1.x) < epsVertex)
{
    // calculates circumcenter coordinates
    // vertical line, calculates mean y and x by intersection
    *yc = (p.y + p1.y)/2;
    xm1 = (p1.x + p2.x)/2;
    ym1 = (p1.y + p2.y)/2;
    m1 = (p2.y - p1.y)/(p2.x - p1.x);
    *xc = -(*yc - ym1)*m1 + xm1;
}
else
// verifies if line p1,p2 is horizontal, then circumcenter is defined by constant line
if(abs(p1.y - p2.y) < epsVertex)
{
    // calculates circumcenter coordinates
    // vertical line, calculates mean x and y by intersection
    *xc = (p1.x + p2.x)/2;
    xm2 = (p2.x + p.x)/2;
    ym2 = (p2.y + p.y)/2;
    m2 = (p.y - p2.y)/(p.x - p2.x);
    *yc = -(*xc - xm2)/m2 + ym2;
}
else
// verifies if line p2,p is horizontal, then circumcenter is defined by constant line
if(abs(p2.y - p.y) < epsVertex)
{
    // calculates circumcenter coordinates
    // vertical line, calculates mean x and y by intersection
    *xc = (p2.x + p.x)/2;
```

```
        xm1 = (p1.x + p2.x)/2;
        ym1 = (p1.y + p2.y)/2;
        m1 = (p2.y - p1.y)/(p2.x - p1.x);
        *yc = -(*xc - xm1)/m1 + ym1;
    }
    else
    // verifies if line p,p1 is horizontal, then circumcenter is defined by constant line
    if(abs(p.y - p1.y) < epsVertex)
    {
        // calculates circumcenter coordinates
        // vertical line, calculates mean x and y by intersection
        *xc = (p.x + p1.x)/2;
        xm1 = (p1.x + p2.x)/2;
        ym1 = (p1.y + p2.y)/2;
        m1 = (p2.y - p1.y)/(p2.x - p1.x);
        *yc = -(*xc - xm1)/m1 + ym1;
    }
    else
    {
        // if both lines are not horizontal/vertical
        xm1 = (p1.x + p2.x)/2;
        ym1 = (p1.y + p2.y)/2;
        xm2 = (p2.x + p.x)/2;
        ym2 = (p2.y + p.y)/2;
        m1 = (p2.y - p1.y)/(p2.x - p1.x);
        m2 = (p.y - p2.y)/(p.x - p2.x);
        *xc = (xm2/m2 + ym2 - xm1/m1 - ym1)/(1/m2 - 1/m1);
        *yc = -(*xc - xm1)/m1 + ym1;
    }

    // calculates radius
    dx = p.x - *xc;
    dy = p.y - *yc;
    *r = sqrt(dx*dx + dy*dy);

    return;
}
```

ANNEX F - UTILITY FUNCTIONS

```
// verifies if point coordinates are defined
bool point_not_undef(struct POINT p)
{
    if(p.x != UNDEF && p.y != UNDEF)
        return(true);
    else
        return(false);
}

// returns if 2 points have the same coordinates
bool points_are_equal(struct POINT p1, struct POINT p2)
{
    return(abs(p1.x - p2.x) < epsVertex && abs(p1.y - p2.y) < epsVertex);
}

// return if 2 edges have the same points
bool edges_are_equal(struct POINT p1, struct POINT p2, struct POINT p3, struct POINT p4)
{
    return((points_are_equal(p1, p3) == true && points_are_equal(p2, p4) == true) ||
           (points_are_equal(p1, p4) == true && points_are_equal(p2, p3) == true));
}

// verifies if points p1,p2 are an edge of a triangle adjacent to the parent triangle and
    returns the pointer to the adjacent triangle
//
// in addition, stores as pointers in a vector the coordinates of the point opposite to the edge
     and the points of the edge, in this order
struct TRIANGLE *adjacent_triangle_by_parent(struct TRIANGLE *parent, struct TRIANGLE *child,
    struct POINT p1, struct POINT p2, struct POINT *point_seq)
{
    struct TRIANGLE *t;
    int i, j;

    // process edges
    for(i = 0;i < 3;i++)
    {
        // adjacent triangle by edge
        t = parent->adjacent[i];

        // if adjacent triangle exists
        if(t != NULL)
        {
            // if points form an edge
            if(points_are_equal(p1, t->p_i) == true && points_are_equal(p2, t->p_j) == true)
            {
                // store points: opposite and from the edge
                point_seq[0] = t->p_k;
                point_seq[1] = t->p_i;
                point_seq[2] = t->p_j;

                // if necessary to change adjacency
                if(parent != child)
                    // define adjacent to adjacent triangle as child triangle
                    for(j = 0;j < 3;j++)
                        if(parent->adjacent[i]->adjacent[j] == parent)
                            parent->adjacent[i]->adjacent[j] = child;

                return(t);
            }
            else
            // if points form an edge
            if(points_are_equal(p1, t->p_j) == true && points_are_equal(p2, t->p_i) == true)
            {
                // store points: opposite and from the edge
                point_seq[0] = t->p_k;
                point_seq[1] = t->p_j;
                point_seq[2] = t->p_i;

                // if necessary to change adjacency
                if(parent != child)
                    // define adjacent to adjacent triangle as child triangle
                    for(j = 0;j < 3;j++)
                        if(parent->adjacent[i]->adjacent[j] == parent)
                            parent->adjacent[i]->adjacent[j] = child;
```

```
            return(t);
        }
        else
        // if points form an edge
        if(points_are_equal(p1, t->p_j) == true && points_are_equal(p2, t->p_k) == true)
        {
            // store points: opposite and from the edge
            point_seq[0] = t->p_i;
            point_seq[1] = t->p_j;
            point_seq[2] = t->p_k;

            // if necessary to change adjacency
            if(parent != child)
                // define adjacent to adjacent triangle as child triangle
                for(j = 0;j < 3;j++)
                    if(parent->adjacent[i]->adjacent[j] == parent)
                        parent->adjacent[i]->adjacent[j] = child;

            return(t);
        }
        else
        // if points form an edge
        if(points_are_equal(p1, t->p_k) == true && points_are_equal(p2, t->p_j) == true)
        {
            // store points: opposite and from the edge
            point_seq[0] = t->p_i;
            point_seq[1] = t->p_k;
            point_seq[2] = t->p_j;

            // if necessary to change adjacency
            if(parent != child)
                // define adjacent to adjacent triangle as child triangle
                for(j = 0;j < 3;j++)
                    if(parent->adjacent[i]->adjacent[j] == parent)
                        parent->adjacent[i]->adjacent[j] = child;

            return(t);
        }
        else
        // if points form an edge
        if(points_are_equal(p1, t->p_k) == true && points_are_equal(p2, t->p_i) == true)
        {
            // store points: opposite and from the edge
            point_seq[0] = t->p_j;
            point_seq[1] = t->p_k;
            point_seq[2] = t->p_i;

            // if necessary to change adjacency
            if(parent != child)
                // define adjacent to adjacent triangle as child triangle
                for(j = 0;j < 3;j++)
                    if(parent->adjacent[i]->adjacent[j] == parent)
                        parent->adjacent[i]->adjacent[j] = child;

            return(t);
        }
        else
        // if points form an edge
        if(points_are_equal(p1, t->p_i) == true && points_are_equal(p2, t->p_k) == true)
        {
            // store points: opposite and from the edge
            point_seq[0] = t->p_j;
            point_seq[1] = t->p_i;
            point_seq[2] = t->p_k;

            // if necessary to change adjacency
            if(parent != child)
                // define adjacent to adjacent triangle as child triangle
                for(j = 0;j < 3;j++)
                    if(parent->adjacent[i]->adjacent[j] == parent)
                        parent->adjacent[i]->adjacent[j] = child;

            return(t);
        }
    }
}
```

```
    // no adjacent triangle by the given edge
    t = NULL;

    return(t);
}

// verifies if points p1,p2 form an edge of triangle t and return the opposite point
struct POINT opposite_point_by_edge(struct POINT p1, struct POINT p2, struct TRIANGLE *t)
{
    struct POINT opposite_point;

    // if triangle exists
    if(t != NULL)
    {
        // verifies edge i,j
        if(edges_are_equal(p1, p2, t->p_i, t->p_j) == true)
        {
            opposite_point = t->p_k;

            return(opposite_point);
        }
        // verifies edge j,k
        if(edges_are_equal(p1, p2, t->p_j, t->p_k) == true)
        {
            opposite_point = t->p_i;

            return(opposite_point);
        }
        // verifies edge i,k
        if(edges_are_equal(p1, p2, t->p_i, t->p_k) == true)
        {
            opposite_point = t->p_j;

            return(opposite_point);
        }
    }

    // set opposite point with undefined coordinates as points p1,p2 are not an edge of triangle
        t
    opposite_point.x = UNDEF;
    opposite_point.y = UNDEF;
    opposite_point.z = UNDEF;
    opposite_point.address = NULL;

    return(opposite_point);
}
```