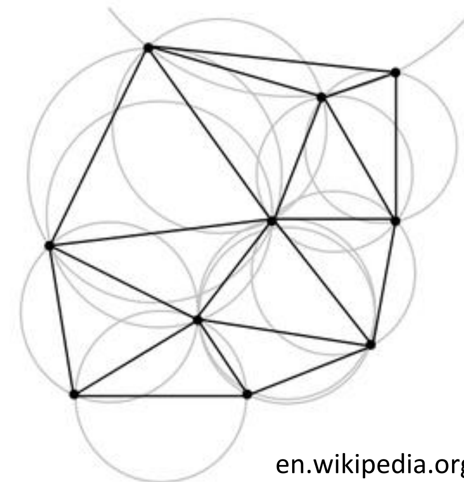
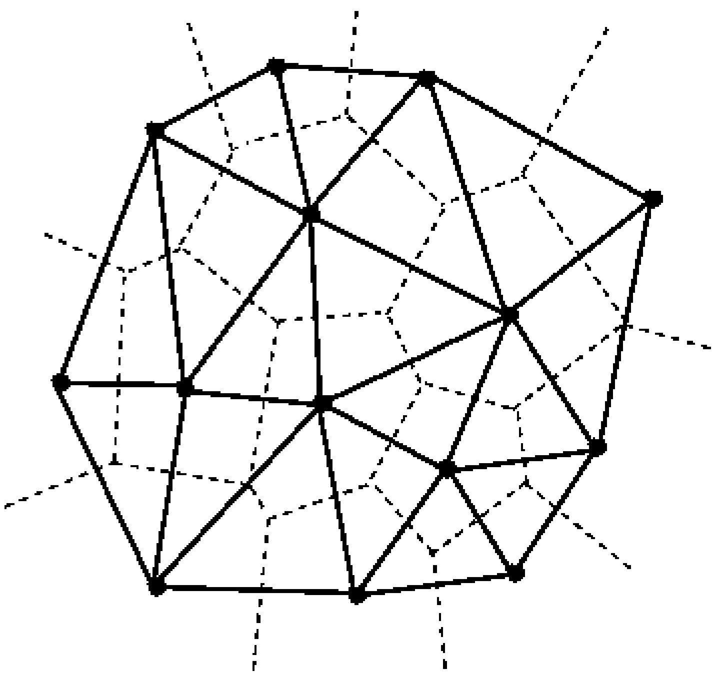
yagosweb.blogspot.com



en.wikipedia.org

Delaunay triangulation



www.comp.lancs.ac.uk

jonathanpuckey.com

Delaunay triangulation

- Triangulation aiming to preserve the triangles to be as equilateral as possible (in such a representation, each triangle represents the local value on the surface in the best way)
- It is unique
  - Independent on the starting point or the orientation of the input dataset
  - If 4 and more points are not lying on a circle

Delaunay triangulation

- Input: $P = \{p_1, p_2, \ldots, p_n\}$
- Output: Triangulation T for P
- Definition of triangulation T for P represents the space division into the set of m triangles $T = \{t_1, t_2, \ldots, t_m\}$ which fulfill:
  - Two arbitrary triangles can share maximally one edge
  - The union of all triangles from T forms the convex hull of P
  - None of the triangles contains another point from P

Active Edge List (AEL)

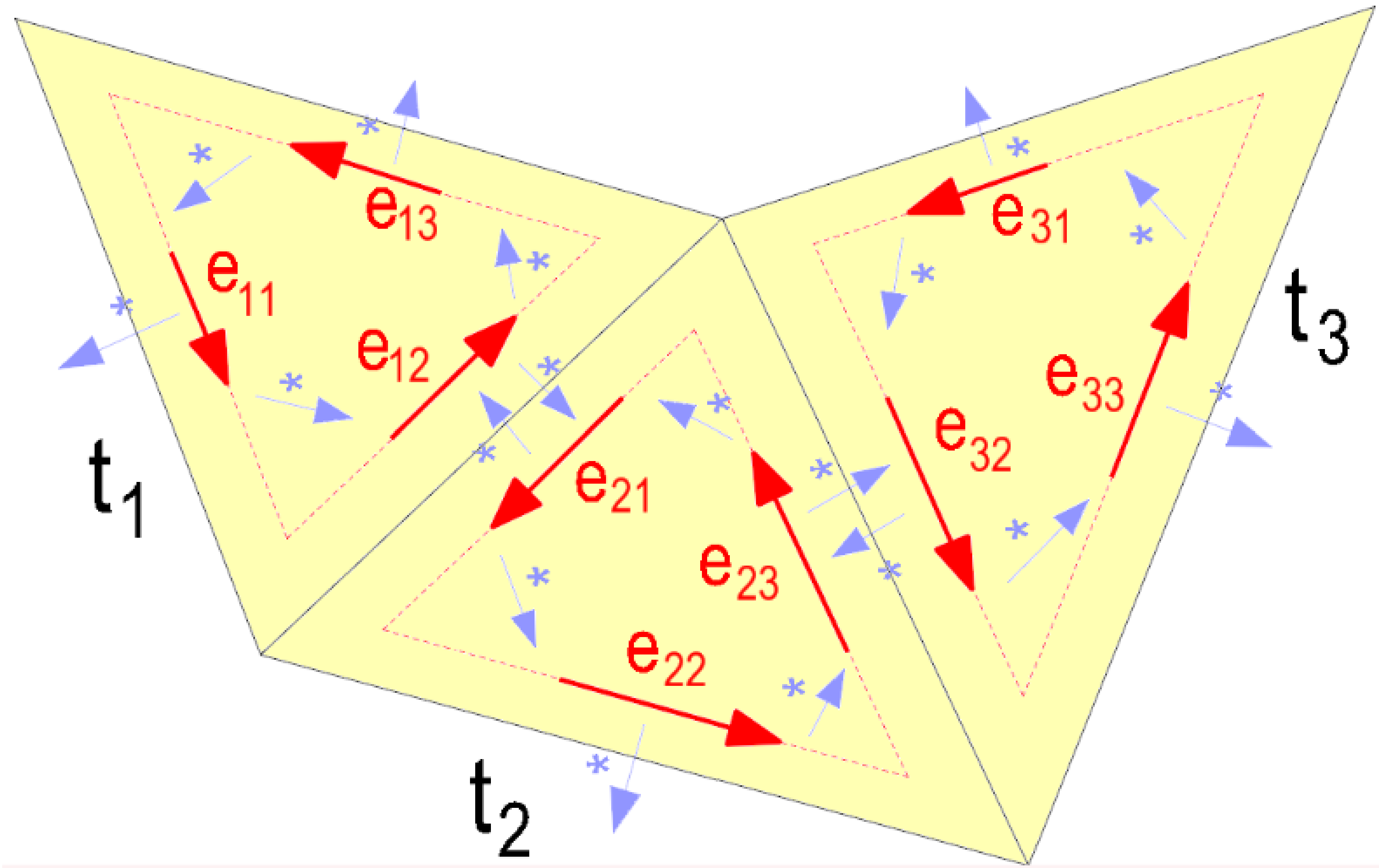
- Data structure often used for construction of DT
- Contains the topology of the DT triangles
- Lets consider two adjacent triangles t_i, t_j from DT, sharing one edge marked as e_{ij} in t_i and as e_{ji} in t_j
- Each edge e_{ij} (Active Edge) in t_i triangle oriented counter-clockwise keeps:
  – Pointer to the following edge e_{i+1} in t_i
  – Pointer to edge e_{ji} from the adjacent triangle t_j

Active Edge List (AEL)

- Except for edges lying on the convex hull *H*, each edge *e* from DT is represented twice (as e_{ij} and e_{ji}), with different orientations
- These doubled edges are called **twin edges**

- Each triangle is then described by a triplet of edges (e_{ij}, e_{i+1j}, e_{i+2j}) with counter-clockwise orientation and forming a Circular List
- The list of all such edges forms the **Active Edge List**

Active Edge List (AEL)



VD construction – algorithms

- Direct construction:
  - Local switching
  - Incremental approach
  - Divide and conquer
- Indirect construction:
  - Via Voronoi diagram

Local switching

- Modifying of a general triangulation to DT
- Based on switching the “illegal” edges in adjacent triangles forming a convex quad

- Complexity $O(n^2)$

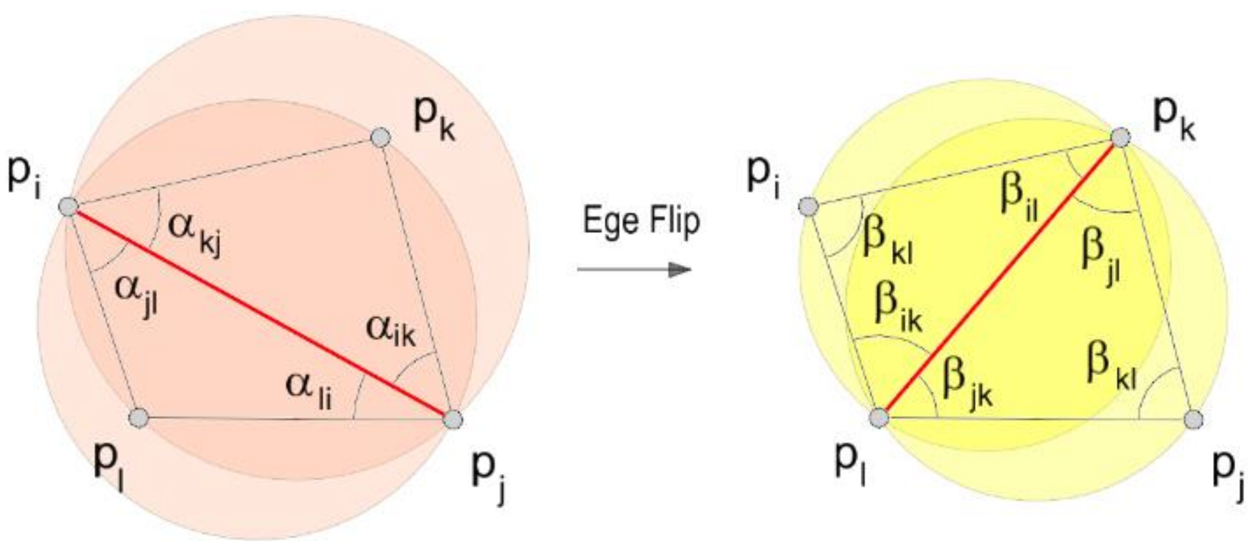
Local switching

Algorithm: **Delaunay Triangulation Local(*P*)**

1. Create some triangulation *T(P)*
2. legal = false;
3. while *T(P)* !legal
4. legal = true;
5. Repeat for each e_i in *T(P)*
6. Take edge e_i and find its incident triangles t_1 and t_2
7. If the union of t_1 and t_2 is convex and illegal
8. **Legalize (t_1, t_2);**
9. legal = false;

Edge legalization

- **Edge flip** = swapping the quad diagonals
- The resulted triangles are both **legal** = locally optimal according to the selected criterion



Edge legalization

- Typical criteria:
  - Minimization of the maximal angle
  - Vertices lying inside a circumscribed circle of the triangle
  - Minimal/maximal triangle height *v*
  - Minimal/maximal area of triangle *S*
  - …

Incremental approach

- Can be used in 2D and 3D
- Incremental addition of points into already created DT
- For already existing Delaunay edge $e = p_1p_2$ we search for such a point p, which has the **minimal** Delaunay distance $d_D(p_1p_2\ , p)$ from p_1p_2
- Each Delaunay edge is oriented, the point p is searched only on the left side from this edge
- We use the test for orientation of the triangle vertices if it is counter-clockwise (determinant test)

Incremental approach

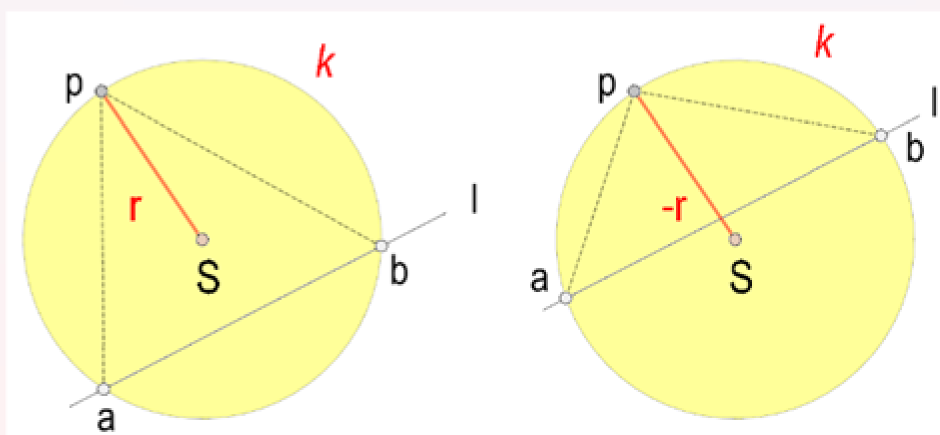
- We add edges of triangle (p_1 , p_2 , p) to DT
- If such a point p does not exist (the examined edge lies on the convex hull), we change the edge orientation and repeat the search

- Complexity $O(n^2)$

Delaunay distance

- Let *k(S, r)* be a circle and *l* a line intersecting with *k* in points *a, b* and *p* point lying on *k*
- Delaunay distance of point *p* from edge *a,b* is marked as $d_D(h, p)$

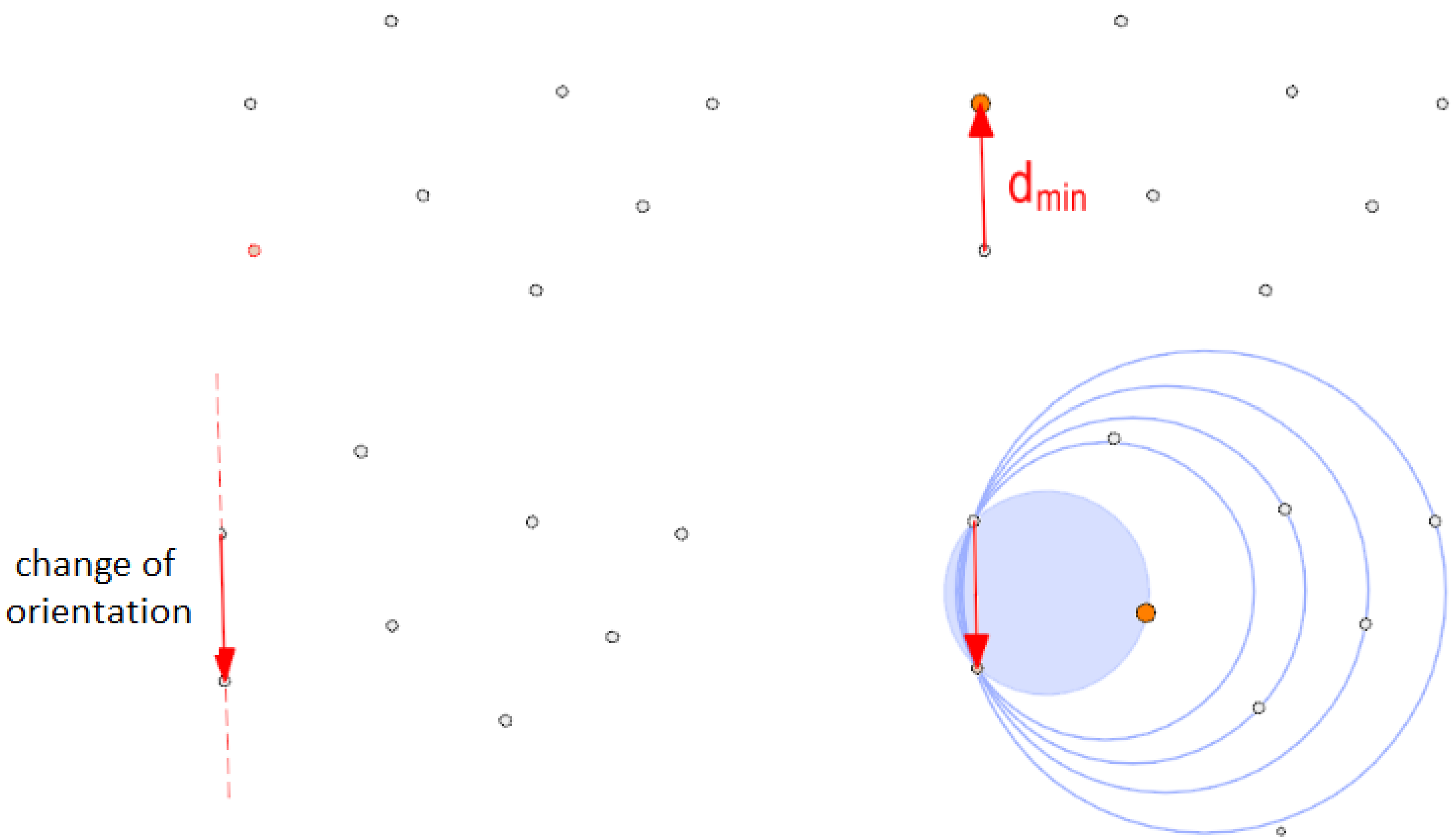
$$d_D(h,p) = \begin{cases} -r & \text{Points } S, p \text{ are in the opposite halfplane wrt. } l \\ r & \text{Points } S, p \text{ are in the same halfplane wrt. } l \end{cases}$$



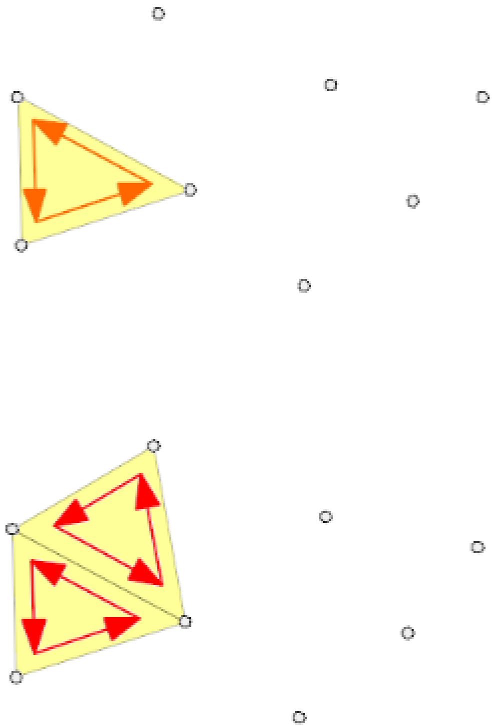
Incremental approach

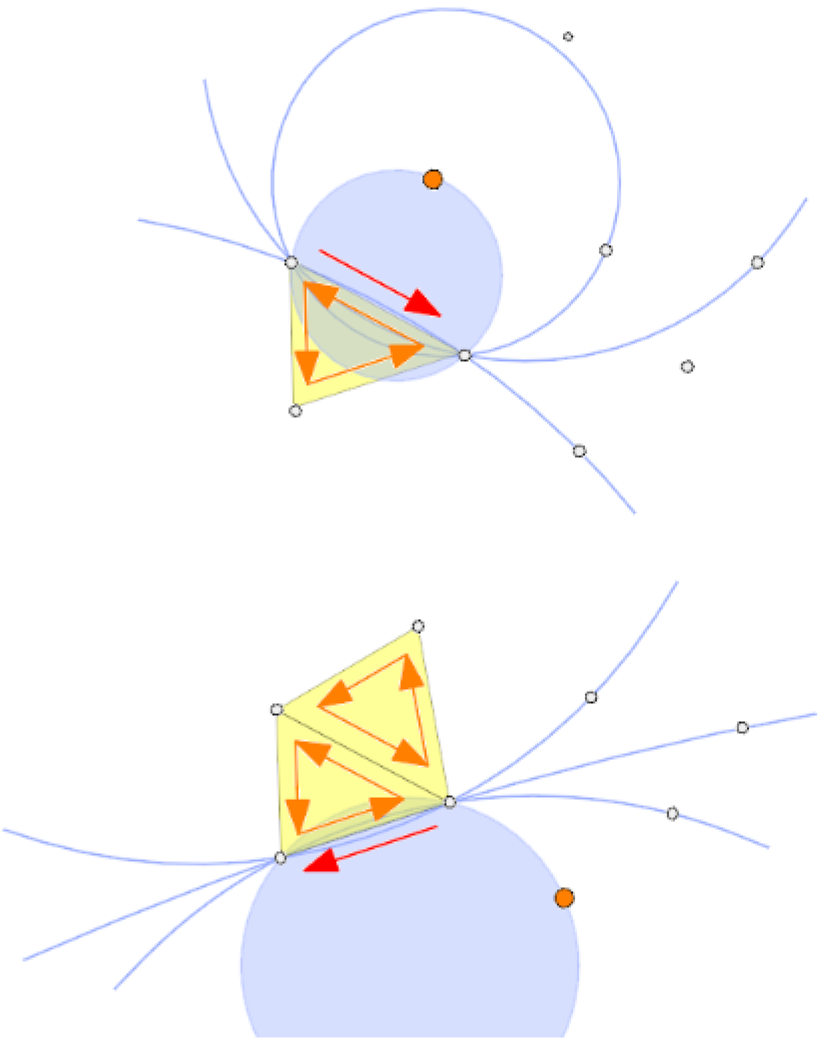
- When constructing we can use the modified AEL structure:
  - It contains edges *e* for whose we are searching for points *p*, it doesn’t store the topology model

Incremental approach

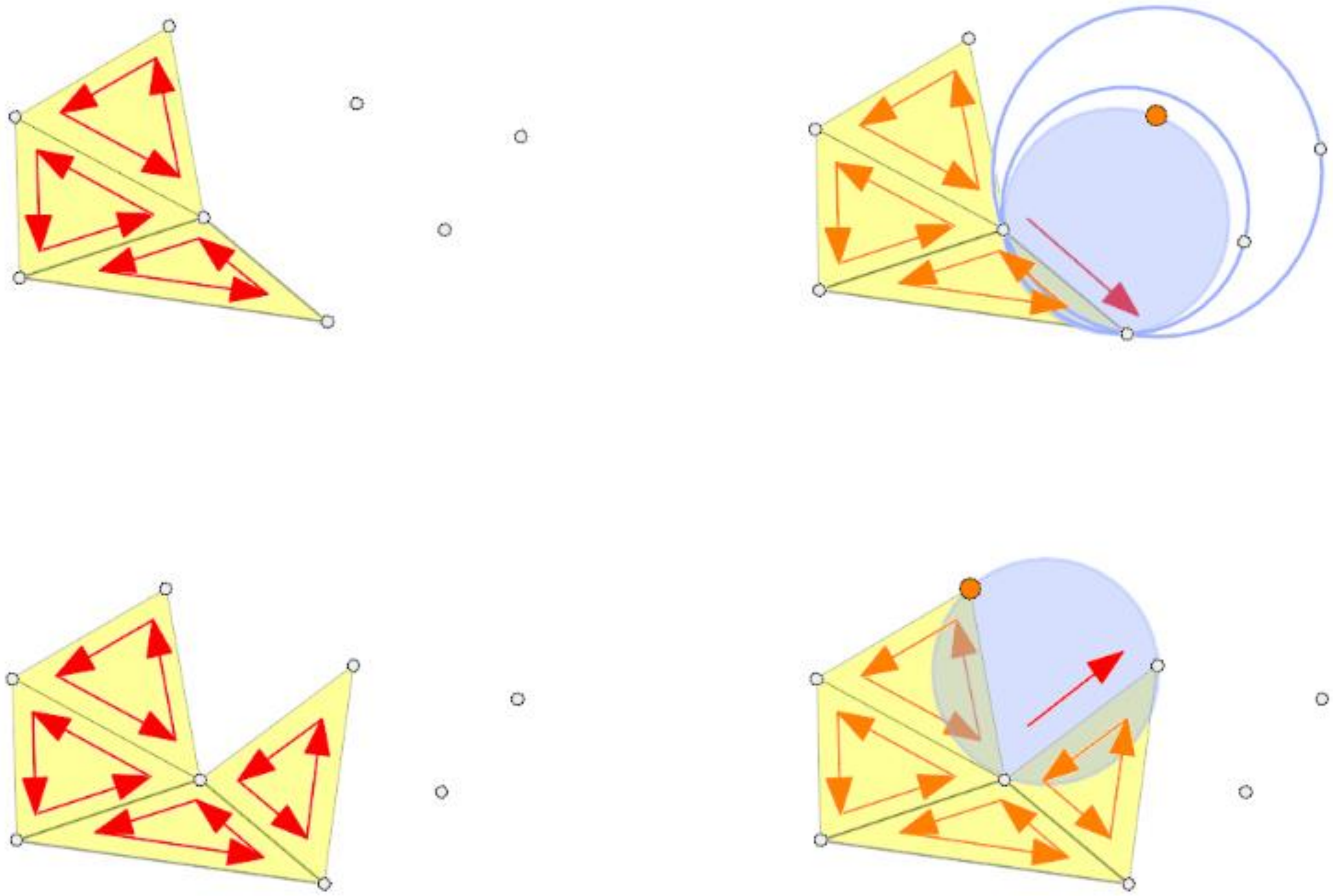


Incremental approach





Incremental approach



Pseudocode

Algorithm: **Delaunay Triangulation Incremental (*S, AEL, DT*)**

1. p_1 = random point from *P*, p2 = the closest point to p_1
2. create edge $e = p_1p_2$;
3. $p = d_D(e)$, point with the smallest Delaunay distance left from e
4. if $p = NULL$, swap orientation $e = p_1p_2$ to $e = p_2p_1$ **and go back to 3**
5. $e_2 = p_2p, e_3 = pp_1$
6. add e, e_2, e_3 to AEL
7. while AEL not empty do
8. $e = p_1p_2$ first edge from AEL
9.   swap orientation $e = p_1p_2$ to $e = p_2p_1$
10.   point p with the smallest Delaunay distance $d_D(e)$ left from e
11.   if p != $NULL$
12. $e_2 = p_2p, e_3 = pp_1$
13.   add e_2, e_3 to AEL (if these or their flips are not in AEL or DT)
14.   Add e to DT
15.   pop (e)

Pseudocode

Algorithm for adding edge *e* to AEL checks if AEL already contains the pair *e′* with opposite orientation.

If so, *e* is removed from AEL.

If not, *e* is added to AEL.

Edge *e* is in both cases added to DT.

The triangulation is stored triangle by triangle.

Algorithm: **Add (*e = ab, AEL, DT*)**

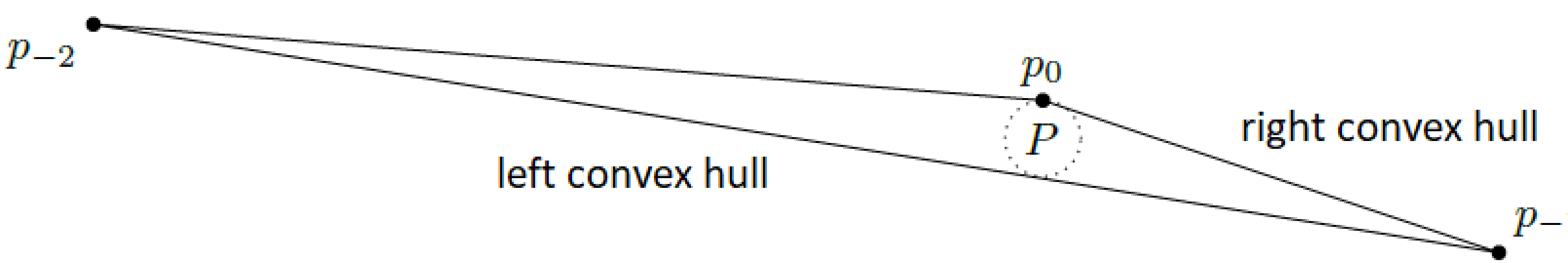
1. create edge *e′ = ba*
2. if (*e′* is in AEL)
3.   remove *ab* from AEL
4. else
5.   push *ab* to AEL
6. push *ab* to DT

Incremental insertion method

- Uses so-called simplex (bounding triangle)
- Frequent method for DT construction
- Complexity $O(n^2)$
- Principle:
  - In each step we add one point to DT and perform the legalization of DT

Incremental insertion method

- Input: set $P = \{p_0, p_1, \ldots, p_n\}$ of points in a plane
- Select p_0 as a point with the highest y-axis value (or also the x-axis)
- We add two other points p_{-1} (sufficiently low and far away to the right) and p_{-2} (sufficiently high and far away to the left) so that P lies inside the triangle $p_0\, p_{-1}\, p_{-2}$



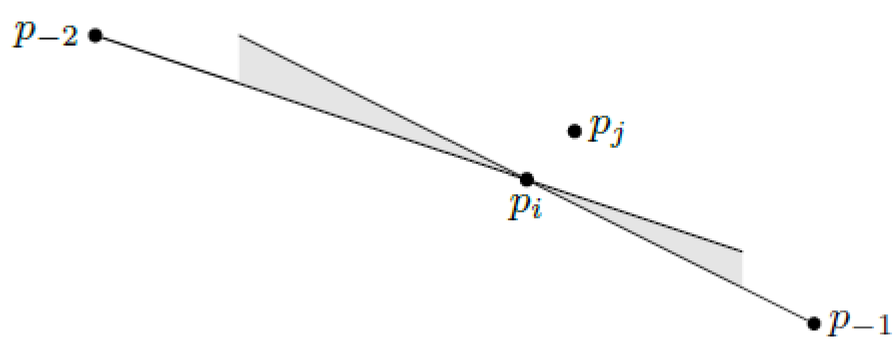
Incremental insertion method

- We create the DT sets $\{p_{-2}, p_{-1}, p_0, p_1, \ldots, p_n\}$ and at the end we remove all edges containing points p_{-2} and p_{-1}
- DT for the set $\{p_{-2}, p_{-1}, p_0\}$ is the triangle $\{p_{-2}, p_{-1}, p_0\}$

Incremental insertion method

- We don't want to determine the exact position of p_{-2}, p_{-1}, so for determining the position of p_j wrt. the oriented line we use the following equivalence:

1. p_j lies on the left side from $p_i p_{-1}$
2. p_j lies on the left side from $p_{-2} p_i$
3. $p_j > p_i$ in a lexicographic order according to y-axis and then to x-axis



Algorithm DELAUNAYTRIANGULATION(P)

Input. A set P of $n+1$ points in the plane.

Output. A Delaunay triangulation of P.

Let p_0 be the lexicographically highest point of P, that is, the rightmost among the points with largest y-coordinate.

Let p_{-1} and p_{-2} be two points in $\mathbb{R}^2$ sufficiently far away and such that P is contained in the triangle $p_0 p_{-1} p_{-2}$.

Initialize $\mathcal{T}$ as the triangulation consisting of the single triangle $p_0 p_{-1} p_{-2}$.

Compute a random permutation $p_1, p_2, \dots, p_n$ of $P \setminus \{p_0\}$.

for $r \leftarrow 1$ **to** n

    do (∗ Insert p_r into $\mathcal{T}$: ∗)

        Find a triangle $p_i p_j p_k \in \mathcal{T}$ containing p_r.

        if p_r lies in the interior of the triangle $p_i p_j p_k$

          then Add edges from p_r to the three vertices of $p_i p_j p_k$, thereby splitting $p_i p_j p_k$ into three triangles.

              LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)

              LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)

              LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)

          else (∗ p_r lies on an edge of $p_i p_j p_k$, say the edge $\overline{p_i p_j}$ ∗)

              Add edges from p_r to p_k and to the third vertex p_l of the other triangle that is incident to $\overline{p_i p_j}$, thereby splitting the two triangles incident to $\overline{p_i p_j}$ into four triangles.

              LEGALIZEEDGE($p_r, \overline{p_i p_l}, \mathcal{T}$)

              LEGALIZEEDGE($p_r, \overline{p_l p_j}, \mathcal{T}$)

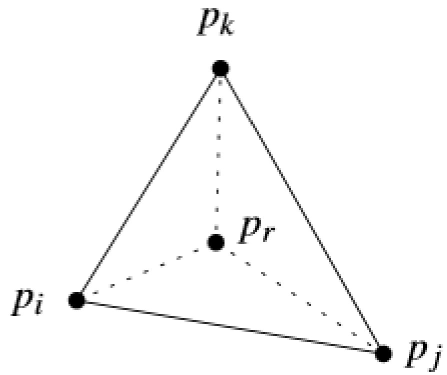
              LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)

              LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)

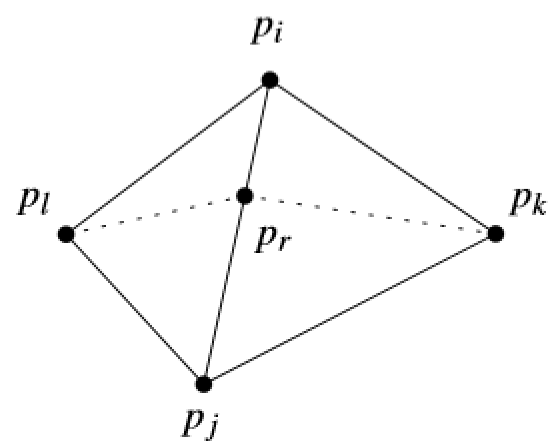
Discard p_{-1} and p_{-2} with all their incident edges from $\mathcal{T}$.

return $\mathcal{T}$

p_r lies in the interior of a triangle



p_r falls on an edge



LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)

(* The point being inserted is p_r, and $\overline{p_i p_j}$ is the edge of $\mathcal{T}$ that may need to be flipped. *)

if $\overline{p_i p_j}$ is illegal

  then Let $p_i p_j p_k$ be the triangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.

    (* Flip $\overline{p_i p_j}$: *) Replace $\overline{p_i p_j}$ with $\overline{p_r p_k}$.

    LEGALIZEEDGE($p_r, \overline{p_i p_k}, \mathcal{T}$)

    LEGALIZEEDGE($p_r, \overline{p_k p_j}, \mathcal{T}$)

Step 7 – finding the triangle containing *p*

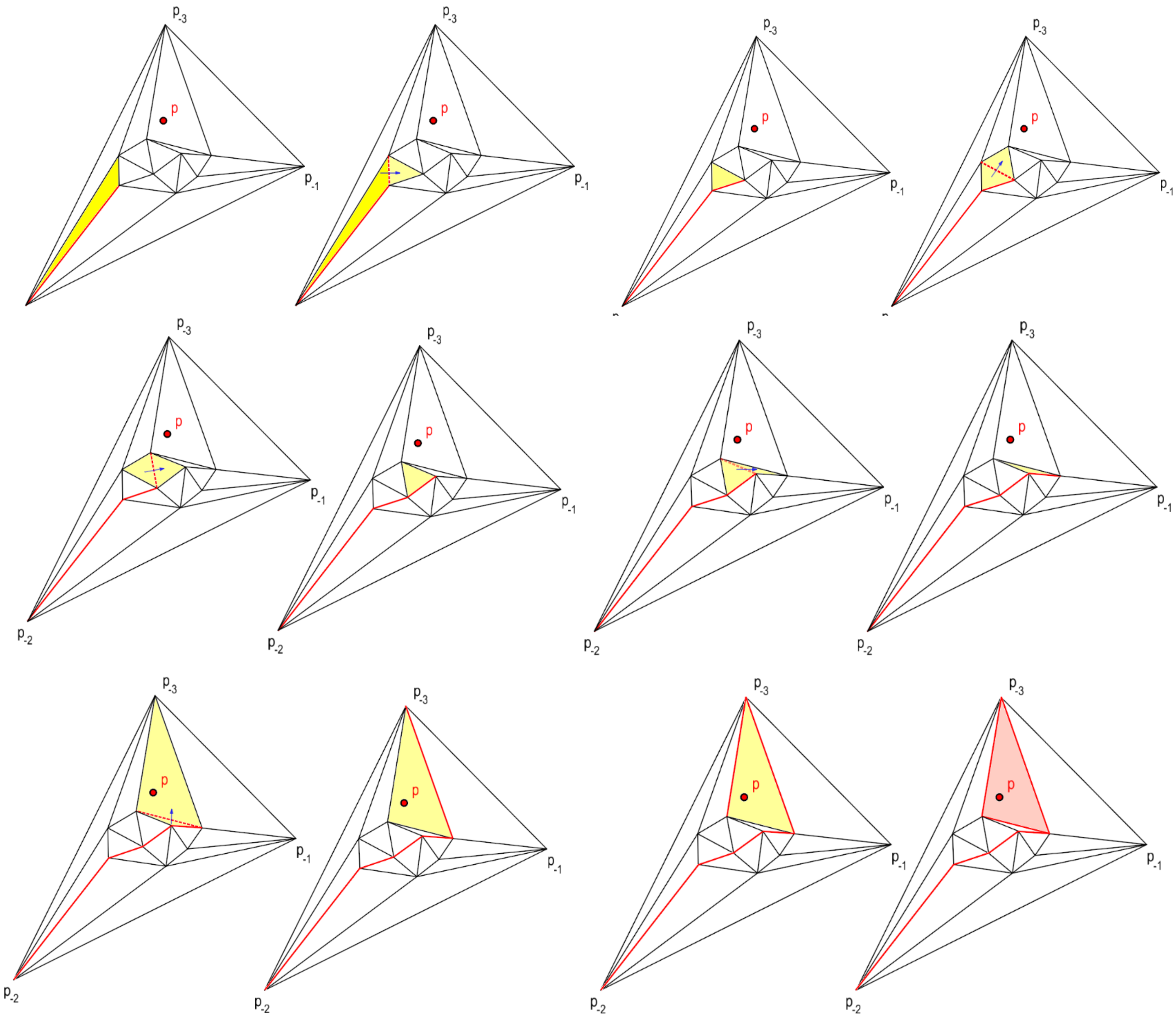
- The most computationally demanding step (it is not efficient to search for *p* in all triangles)
- The most common methods:
  - Walking method (heuristic method, $O(n^2)$)
  - DAG tree (ternary tree construction, $O(n \log n)$)

Walking method

- By traversing the adjacent triangles we are gradually approaching the searched triangle t_i
- We are testing the mutual position of p and edge e_{ij} in AEL.

$$p \begin{cases} \text{on the left side from } e_{i,j} \text{ in } t_i, & \text{we are testing } e_{i+1,j} \text{ in } t_i \\ \text{on the right side from } e_{i,j} \text{ in } t_i, & \text{we are testing } e_{j,i} \text{ in } t_j \end{cases}$$

- Point p lies on the left side from all edges of the searched triangle



p_{-3}
p
p_{-1}
p_{-2}

Divide and conquer

- Input set of points is divided into smaller parts, each of them is triangulated separately
- Resulting triangulations are merged and legalized

Assignment

- Implement the Delaunay triangulation using the incremental approach

Useful details for implementation

- We have to be able to determine the circumscribed circle = circle containing three vertices
- We can do this in the following way:
  - Create a class RealPoint(float x, float y)
    - Its ***distance*** method calculates the distance between points p1 and p2:
      - $sqrt((p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2)$

Useful details for implementation

- Class **Circle** is determined by its center (RealPoint c) and radius (float r)

- Testing if a point *p* lies inside a circle:
  - Method ***inside***
    - if (c.distanceSq(p) < r^2) return true;

      where distanceSq = $(p_1.x - p_2.x)^2 + (p_1.y - p_2.y)^2$

Useful details for implementation

- Calculating the circle with three points lying on it (RealPoint p_1, p_2, p_3):
  - Method ***circumCircle***(p_1, p_2, p_3)

```
cp = crossproduct (p1, p2, p3);
if (cp <> 0) {
        p1Sq = p1.x^2 + p1.y^2;
        p2Sq = p2.x^2 + p2.y^2;
        p3Sq = p3.x^2 + p3.y^2;
        num = p1Sq *(p2.y - p3.y) + p2Sq *(p3.y - p1.y) +
                p3Sq *(p1.y - p2.y);
        cx = num / (2.0 * cp);
        num = p1Sq *(p3.x - p2.x) + p2Sq*(p1.x - p3.x) +
                p3Sq*(p2.x - p1.x);
        cy = num / (2.0f * cp); c.set(cx, cy);
        c.set(cx, cy);
        r = c.distance(p1);
```

Useful details for implementation

- crossproduct (p_1, p_2, p_3)>

  $u_1 = p_2.x() - p_1.x();$

  $v_1 = p_2.y() - p_1.y();$

  $u_2 = p_3.x() - p_1.x();$

  $v_2 = p_3.y() - p_1.y();$

  return $u_1 * v_2 - v_1 * u_2;$