Comp. Appl. Math. (2016) 35:423–438
DOI 10.1007/s40314-014-0198-8

Implementation of linear minimum area enclosing triangle algorithm

Application note

Ovidiu Pârvu · David Gilbert

Received: 22 February 2014 / Revised: 18 September 2014 / Accepted: 15 October 2014 /
Published online: 15 November 2014
© The Author(s) 2014. This article is published with open access at Springerlink.com

Abstract An algorithm which computes the minimum area triangle enclosing a convex polygon in linear time already exists in the literature. The paper describing the algorithm also proves that the provided solution is optimal and a lower complexity sequential algorithm cannot exist. However, only a high-level description of the algorithm was provided, making the implementation difficult to reproduce. The present note aims to contribute to the field by providing a detailed description of the algorithm which is easy to implement and reproduce, and a benchmark comprising 10,000 variable sized, randomly generated convex polygons for illustrating the linearity of the algorithm.

Keywords Minimum area triangle · Benchmark · Convex polygon · Rotating caliper · Computational geometry

Mathematics Subject Classification 68Q25 · 68U05

1 Introduction

The problem addressed by this note is to find the triangle of minimum area enclosing a convex polygon in the Euclidean plane E^2. Similarly, the problem of finding the triangle of minimum area enclosing a set P of points in the Euclidean plane E^2 can be solved using the

---

Communicated by José Mario Martínez.

Electronic supplementary material The online version of this article (doi:10.1007/s40314-014-0198-8) contains supplementary material, which is available to authorized users.

O. Pârvu (✉) · D. Gilbert
Department of Computer Science, Brunel University, Uxbridge, Middlesex UB8 3PH, UK
e-mail: ovidiu.parvu@brunel.ac.uk

D. Gilbert
e-mail: david.gilbert@brunel.ac.uk

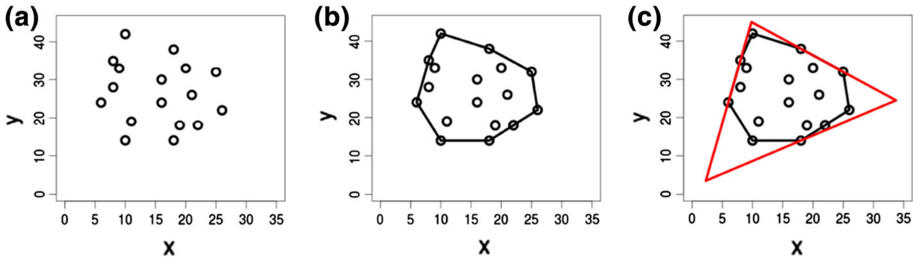

Fig. 1 **a** Set of points in the Euclidean plane E^2, **b** enclosed by their convex hull and **c** by the minimum area triangle

same algorithm by first computing the convex hull enclosing the set of points P using any 2D convex hull algorithm; see Fig. 1 for an example.

The algorithm described in Klee and Laskowski (1985) finds all minimum area triangles enclosing a given convex polygon in $O(n \log^2(n))$. Inspired by Toussaint's rotating callipers procedure (Toussaint 1983) O'Rourke improved the algorithm reducing its complexity to $\Theta(n)$ (O'Rourke et al. 1986). Similarly, the authors of Chandran and Mount (1992) describe a method of parallelising Klee and Laskowski's algorithm such that the minimum area enclosing triangle can be found in $O(\log \log(n))$ using $(n \ / \ \log \log(n))$ processors.

The main contributions of this note are:

- A detailed and reproducible algorithm for computing the minimal area enclosing triangles (Sects. 2 and 4);
- A step by step description of an execution of the algorithm implementation, including a screencast (Sect. 3);
- A benchmark of 10,000 randomly generated convex polygons for assessing the efficiency of the algorithm (Sect. 5);
- Publicly available C++ implementation of the algorithm released both as a standalone software project and as a module of the Computer Vision library OpenCV (Bradski and Kaehler 2008) (Sects. 6 and 7).

This scientific note is organised as follows: we introduce the main algorithm in Sect. 2 and provide a detailed step by step execution of its implementation in Sect. 3. The required subalgorithms are described in Sect. 4, and results of executing the algorithm implementation against a benchmark are discussed in Sect. 5. Methods for verifying the correctness of the implementation are presented in Sect. 6. Finally, concluding remarks are provided in Sect. 7.

2 Main algorithm

Theorems and lemmas underlying the main algorithm can be found in Klee and Laskowski (1985), O'Rourke et al. (1986) and will not be repeated here.

The following notations will be used throughout:

- a, b, c—indices pointing to the polygon vertices in a clockwise order;
- A, B, C—the sides of the current enclosing triangle;
- $vertexA, vertexB, vertexC$—the vertices of the current enclosing triangle;
- $p + 1$—$polygon$ vertex succeeding p considering a clockwise order;
- $p - 1$—$polygon$ vertex preceding p considering a clockwise order;

- $h(p)$—distance of p from line determined by side C;
- $validationFlag$—used to record what validation conditions should be used.

The algorithm for finding minimum area enclosing triangles is based on an elegant geometric characterisation initially introduced in Klee and Laskowski (1985). The algorithm iterates over each edge of the convex polygon setting side C of the enclosing triangle to be flush with this edge. A side S is said to be flush with edge E if $S \supseteq E$. The authors of O'Rourke et al. (1986) prove that for each fixed flush side C a local minimum enclosing triangle exists. Moreover, the authors have shown that:

- The midpoints of the enclosing triangle's sides must touch the polygon.
- There exists a local minimum enclosing triangle with at least two sides flush with edges of the polygon. The third side of the triangle can be either flush with an edge or tangent to the polygon.

Thus, for each flush side C the algorithm will find the second flush side and set the third side either flush/tangent to the polygon.

The main algorithm is described in O'Rourke et al. (1986) using abstract, high-level descriptions. In contrast, we will describe both the main algorithm and all required subalgorithms in great depth and in an intuitive manner.

First of all, the Main Algorithm 1 contains a loop which iterates over each edge of the convex polygon and sets the side C of the triangle flush with the selected edge. A necessary condition for finding a minimum enclosing triangle is that b is on the right chain and a on the left. The first step inside the loop is therefore to move the index b on the right chain using the *AdvanceBToRightChain()* subalgorithm. The initialisation of a was made in such a manner that it is on the left chain already.

The next condition which must be fulfilled is that a and b must be *critical* or *high*. The *MoveAIfLowAndBIfHigh()* subalgorithm advances a and b until this condition is fulfilled.

Next b will be advanced until $[gamma(a)\ b]$ is tangent to the convex polygon via the *SearchForBTangency()* subalgorithm.

Afterwards the subalgorithm *UpdateSidesCA()* computes the vertices defining sides A and C of the enclosing triangle.

If the tangency was not reached in the previous step (see *IsNotBTangency()*) then sides A and B are updated (see *UpdateSidesBA()*). Otherwise only side B needs to be updated (see *UpdateSideB()*).

Finally, if the found enclosing triangle is minimal (see *IsLocalMinimalTriangle()*) and its area is less than the area of the optimal enclosing triangle found so far then the optimal enclosing triangle is updated (see *UpdateMinimumAreaEnclosingTriangle()*).

The main algorithm and subalgorithms/functions *AdvanceBToRightChain()*, *MoveAIfLowAndBIfHigh()*, *SearchForBTangency()*, *IsNotBTangency()*, *UpdateSidesBA()*, *Update SideB()* and *UpdateMinimumAreaEnclosingTriangle()* are partially described in O'Rourke et al. (1986) as well. However, they are merged into a single algorithm, some steps are not explicitly described and all remaining subalgorithms/functions are not given. This note complements O'Rourke et al. (1986) by explicitly describing all subalgorithms required to implement the main algorithm.

Algorithm 1 Algorithm for computing the minimum area enclosing triangle

Require: $polygon$ is convex and contains more than three vertices;
Ensure: $minAreaTriangle$ is a set of three vertices defining the minimum area enclosing triangle; $minArea$ is the area of the minimum area enclosing triangle;

```
Global variables:
    polygon
    a, b, c
    A, B, C
    vertexA, vertexB, vertexC
    validationFlag

1: procedure MinAreaTriangle(polygon, minAreaTriangle, minArea)
2:     a ← 2;
3:     b ← 3;
4:     for c ← 1, nr. of polygon vertices do          ▷ Side C is flush with edge [c, c − 1]
5:         AdvanceBToRightChain();
6:         MoveAIfLowAndBIfHigh();
7:         SearchForBTangency();
8:
9:         UpdateSidesCA();
10:
11:        if IsNotBTangency() then
12:            UpdateSidesBA();
13:        else
14:            UpdateSideB();
15:        end if
16:
17:        if IsLocalMinimalTriangle() then
18:            UpdateMinimumAreaEnclosingTriangle(minAreaTriangle, minArea);
19:        end if
20:    end for
21: end procedure
```

3 Simple usage example

A simple usage example was chosen to illustrate the way in which minimum enclosing triangles of convex polygons are computed. The considered convex polygon was defined by the following set of points:

- $P(300, 700)$;
- $Q(400, 480)$;
- $R(643, 200)$;
- $S(800, 1100)$;
- $T(1202, 1005)$.

In order to capture the execution of each step of the main algorithm a screencast was recorded. The screen was split up in two halves as shown in Fig. 2. The left half of the screen shows the current active line of code. Conversely, the right half of the screen illustrates the current progress of the algorithm as an image. For all execution steps this image contains the convex polygon and the points a, b and c. At particular execution steps the points γ_a, γ_b or the minimum enclosing triangle are displayed as well.

The initial position of the points a, b and c, and the minimum enclosing triangles computed after each iteration of the for loop in the main algorithm are depicted in Fig. 3.

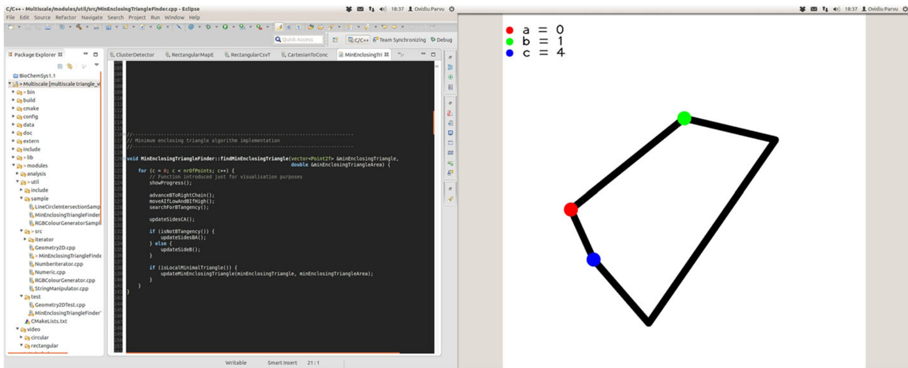

Fig. 2 Step by step execution of the minimum enclosing triangle algorithm implementation. The *left side* of the image shows the currently executed step of the main algorithm. The *right side* of the image illustrates the polygon and the points *a*, *b* and *c*

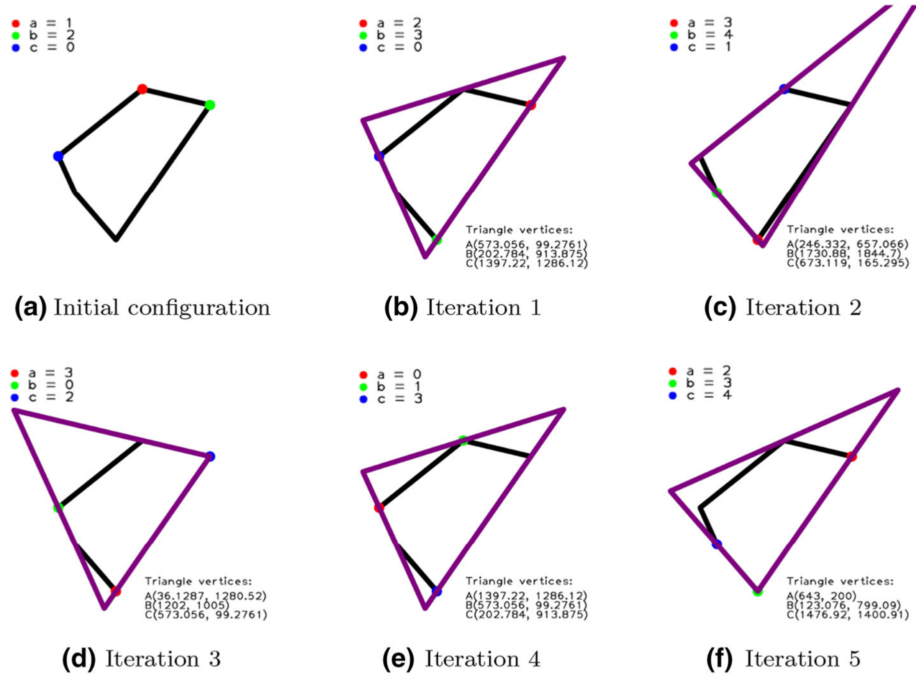

Fig. 3 **a** Initial position of the points *a*, *b* and *c*. **b–f** The minimum enclosing triangle computed at each iteration of the for loop in the main algorithm

The screencast embedded with detailed audio explanations is available as a video at http://people.brunel.ac.uk/~cspgoop/data/notes/2014/min_enclosing_triangle.

4 Subalgorithms

The semantics of the *AdvanceBToRightChain()*, *MoveAIfLowAndBIfHigh()*, *SearchForBTangency()*, *UpdateSidesCA()*, *IsNotBTangency()*, *UpdateSidesBA()*, *UpdateSideB()*, *IsLo-*

calMinimalTriangle() and *UpdateMinimumAreaEnclosingTriangle()* subalgorithms/functions was given in Sect. 2 and is not restated here.

Algorithm 2 Algorithm for advancing b to the right chain

```
1: procedure ADVANCEBTORIGHTCHAIN
2:     while h(b + 1) ≥ h(b) do
3:         b ← b + 1;
4:     end while
5: end procedure
```

Algorithm 3 Algorithm for advancing a if the edge $[a, a - 1]$ is low, and advancing b if the edge $[b, b - 1]$ is high

```
1: procedure MOVEAIFLOWANDBIFHIGH
2:     while h(b) > h(a) do
3:         if Gamma(a, gammaOfA) AND IntersectsBelow(gammaOfA, b) then
4:             b ← b + 1;
5:         else
6:             a ← a + 1;
7:         end if
8:     end while
9: end procedure
```

Algorithm 4 Search for the tangency of side B

```
1: procedure SEARCHFORBTANGENCY
2:     while (Gamma(b, gammaOfB) AND IntersectsBelow(gammaOfB, b)) AND
3:             (h(b) ≥ h(a − 1)) do
4:         b ← b + 1;
5:     end while
6: end procedure
```

Algorithm 5 Update the sides C and A

```
1: procedure UPDATESIDESCA
2:     C.StartVertex ← polygon[c − 1];
3:     C.EndVertex ← polygon[c];
4:
5:     A.StartVertex ← polygon[a − 1];
6:     A.EndVertex ← polygon[a];
7: end procedure
```

Algorithm 6 Check if the B side tangency has not been achieved

```
1: function IsNotBTangency
2:     if (Gamma(b, gammaOfB) AND IntersectsAbove(gammaOfB, b)) OR
3:         (h(b) < h(a − 1)) then
4:         return true;
5:     else
6:         return false;
7:     end if
8: end function
```

Algorithm 7 Update the sides B and A

```
1: procedure UpdateSidesBA
2:     B.StartVertex ← polygon[b − 1];
3:     B.EndVertex ← polygon[b];
4:
5:     if MiddlePointOfSideB(sideBMiddlePoint) AND
6:         (h(sideBMiddlePoint) < h(a − 1)) then
7:         A.StartVertex ← polygon[a − 1];
8:         A.EndVertex ← FindVertexCOnSideB();
9:
10:        validationFlag ← SIDE_A_TANGENT;          ▷ Constant
11:    else
12:        validationFlag ← SIDES_FLUSH;             ▷ Constant
13:    end if
14: end procedure
```

Algorithm 8 Update the side B

```
1: procedure UpdateSideB
2:     Gamma(b, B.StartVertex);
3:     B.EndVertex ← polygon[b];
4:
5:     validationFlag ← SIDE_B_TANGENT;              ▷ Constant
6: end procedure
```

In order to validate every enclosing triangle Algorithm 9 checks if the sides A, B and C determined by Algorithm 1 intersect and if their midpoints touch the polygon.

Algorithm 9 Check if the obtained enclosing triangle is a local minimum

```
1: function IsLocalMinimalTriangle
2:     if (A and B do not intersect) OR (A and C do not intersect) OR
3:         (B and C do not intersect) then
4:         return false;
5:     else
6:         vertexA ← intersection of B and C;
7:         vertexB ← intersection of A and C;
8:         vertexC ← intersection of A and B;
9:
10:        return IsValidMinimalTriangle();
11:    end if
12: end function
```

Algorithm 10 Check if the obtained enclosing triangle is a valid local minimum

```
1: function IsValidMinimalTriangle
2:     midpointA = middle point between vertexB and vertexC;
3:     midpointB = middle point between vertexA and vertexC;
4:     midpointC = middle point between vertexA and vertexB;
5:
6:     if validationFlag == SIDE_A_TANGENT then
7:         validA ← (midpointA == polygon[a − 1]);
8:     else
9:         validA ← (midpointA ∈ line segment [A.StartVertex, A.EndVertex]);
10:    end if
11:
12:    if validationFlag == SIDE_B_TANGENT then
13:        validB ← (midpointB == polygon[b]);
14:    else
15:        validB ← (midpointB ∈ line segment [B.StartVertex, B.EndVertex]);
16:    end if
17:
18:    validC ← (midpointC ∈ line segment [C.StartVertex, C.EndVertex]);
19:
20:    return (validA AND validB AND validC);
21: end function
```

Algorithm 11 Check if the middle point of side B exists and find it if it does

```
1: function MiddlePointOfSideB(middlePoint)
2:     if (B and C do not intersect) OR (B and A do not intersect) then
3:         return false;
4:     end if
5:
6:     vertexA ← intersection of B and C;
7:     vertexC ← intersection of A and B;
8:
9:     middlePoint ← middle point of vertexA and vertexC;
10:
11:    return true;
12: end function
```

Definition 1 The line L determined by $gammaPoint$ and $polygon[index]$ intersects the polygon *below* $polygon[index]$ if the following conditions hold:

1. The *ray* $[polygon[index]\ gammaPoint')$ intersects the polygon, where $gammaPoint'$ is a point on L such that $polygon[index]$ is the middle point of the line segment $[gammaPoint\ gammaPoint']$. Let us denote the intersection point of the ray with the polygon, if it exists, as $intersectionPoint$;
2. $h(intersectionPoint) < h(polygon[index])$.

Erroneous triangles will be obtained if the *line* $(polygon[index]\ gammaPoint')$ instead of the *ray* $[polygon[index]\ gammaPoint')$ is considered when computing the intersection with the polygon; see Klee and Laskowski (1985) for more details why the ray and not the line is considered.

Definition 2 The line L determined by $gammaPoint$ and $polygon[index]$ intersects the polygon *above* $polygon[index]$ if the following conditions hold:

1. The *ray* $[polygon[index]\ gammaPoint)$ intersects the polygon. Let us denote this point, if it exists, as $intersectionPoint$;
2. $h(intersectionPoint) > h(polygon[index])$.

According to condition 1 in Definitions 1 and 2, the intersection of the ray with the polygon has to be computed. However, this would lead to an increase in the overall complexity of the main algorithm. Thus an alternative constant complexity solution which considers only the angle (the slope could be considered as well) of the ray determined by $gammaPoint$ and $polygon[index]$ is employed. In this case the point $gammaPoint'$ is no longer required because the angle of the ray $[polygon[index]\ gammaPoint')$ is equal to the angle of the ray $[gammaPoint\ polygon[index])$.

Both *IntersectsAbove* and *IntersectsBelow* functions call the function *Intersects* which checks if the line intersects the polygon `ABOVE/BELOW` the point $polygon[index]$, or is `CRITICAL`. The only difference between *IntersectsAbove* and *IntersectsBelow* is the way in which they compute the angle of the line determined by $gammaPoint$ and $polygon[index]$ as can be seen in Algorithms 12 and 13. The angle is computed differently because the rays considered in condition 1 (Definitions 1 and 2) differ.

Having received the angle of the line determined by $gammaPoint$ and the point $polygon[index]$ the function $Intersects$ checks using angle comparison (slope comparison could be employed as well) if the line intersects the polygon `ABOVE/BELOW` the point $polygon[index]$ or is `CRITICAL`; see Algorithm 14 for more details.

Algorithm 12 Check if the line determined by $gammaPoint$ and $polygon[index]$ intersects the polygon below $polygon[index]$

```
1: function IntersectsBelow(gammaPoint, index)
2:     // The order of the points is reversed compared to the IntersectsAbove function
3:     angle ← AngleOfLine(polygon[index], gammaPoint);
4:
5:     return (Intersects(angle, index) == BELOW);
6: end function
```

Algorithm 13 Check if the line determined by $gammaPoint$ and $polygon[index]$ intersects the polygon above $polygon[index]$

```
1: function IntersectsAbove(gammaPoint, index)
2:     // The order of the points is reversed compared to the IntersectsBelow function
3:     angle ← AngleOfLine(gammaPoint, polygon[index]);
4:
5:     return (Intersects(angle, index) == ABOVE);
6: end function
```

Algorithm 14 Check if the line determined by $gammaPoint$ and $polygon[index]$ intersects the polygon below/above $polygon[index]$ or is critical

```
1: function INTERSECTS(angle, index)                          ▷ Time complexity: O(1)
2:     anglePred ← AngleOfLine(polygon[index − 1], polygon[index]);
3:     angleSucc ← AngleOfLine(polygon[index + 1], polygon[index]);
4:     angleC ← AngleOfLine(polygon[c − 1], polygon[c]);
5:
6:     if IsAngleBtwPredAndSucc(angleC, anglePred, angleSucc) then
7:         if IsAngleBtwNonReflex(angle, anglePred, angleC) OR
8: 1.5em         (angle == anglePred) then
9:             return IntersectsAboveOrBelow(index − 1, index);
10:        else if IsAngleBtwNonReflex(angle, angleSucc, angleC) OR
11:                (angle == angleSucc) then
12:            return IntersectsAboveOrBelow(index + 1, index);
13:        end if
14:    else
15:        if IsAngleBtwNonReflex(angle, anglePred, angleSucc) OR
16: 1.5em        ((angle == anglePred) AND (angle != angleC)) OR
17: 1.5em        ((angle == angleSucc) AND (angle != angleC)) then
18:            return BELOW;
19:        end if
20:    end if
21:
22:    return CRITICAL;
23: end function
```

Algorithm 15 Check if the line intersects the polygon below/above considering successor/predecessor

```
1: function INTERSECTSABOVEORBELOW(succOrPredIndex, index)
2:     if h(succOrPredIndex) > h(index) then
3:         return ABOVE;
4:     else
5:         return BELOW;
6:     end if
7: end function
```

Algorithm 16 Compute the angle of the line determined by points a and b wrt. Ox axis

```
1: function ANGLEOFLINE(a, b)
2:     y ← b.y − a.y;
3:     x ← b.x − a.x;
4:
5:     angle ← arctangent(y, x) * 180 / π;
6:
7:     if angle < 0 then
8:         return (angle + 360);
9:     else
10:        return angle;
11:    end if
12: end function
```

Algorithm 17 Check if the given angle is between successor or predecessor

```
1: function IsAngleBtwPredAndSucc(angle, anglePred, angleSucc)
2:     if IsAngleBtwNonReflex(angle, anglePred, angleSucc) then
3:         return true;
4:     else if IsOppositeAngleBtwNonReflex(angle, anglePred, angleSucc) then
5:         angle ← OppositeAngle(angle);
6:
7:         return true;
8:     end if
9:
10:    return false;
11: end function
```

Algorithm 18 Check if the given angle is between the non-reflex angle (i.e. < 180 degrees) determined by angles 1 and 2

```
1: function IsAngleBtwNonReflex(angle, angle1, angle2)     ▷ Angles expressed in degrees
2:     if |angle1 - angle2| > 180 then
3:         if angle1 > angle2 then
4:             return ((angle1 < angle) AND (angle ≤ 360)) OR
5:                    ((0 ≤ angle) AND (angle < angle2));
6:         else
7:             return ((angle2 < angle) AND (angle ≤ 360)) OR
8:                    ((0 ≤ angle) AND (angle < angle1));
9:         end if
10:    else
11:        if ((angle1 - angle2) mod 180) > 0 then
12:            return (angle2 < angle) AND (angle < angle1);
13:        else
14:            return (angle1 < angle) AND (angle < angle2);
15:        end if
16:    end if
17: end function
```

Algorithm 19 Check if the opposite angle is between the non-reflex angle (i.e. < 180 degrees) determined by angles 1 and 2

```
1: function IsOppositeAngleBtwNonReflex(angle, angle1, angle2)     ▷ Angles expressed in degrees
2:     oppositeAngle ← OppositeAngle(angle);
3:
4:     return IsAngleBtwNonReflex(oppositeAngle, angle1, angle2);
5: end function
```

Algorithm 20 Return the opposite of an angle

```
1: function OppositeAngle(angle)     ▷ Angle expressed in degrees
2:     if angle > 180 then
3:         return (angle - 180);
4:     else
5:         return (angle + 180);
6:     end if
7: end function
```

The point $\gamma(p)$ (Gamma) is the point on the line $[a, a-1]$ such that $h(\gamma(p)) = 2 \times h(p)$. In order to find $\gamma(p)$ we consider the intersection between lines $L1$, $L2$ and L3, where:

- $L1$: The line $[a, a-1]$;
- $L2, L3$: The lines parallel to and at a distance of $(2 \times h(p))$ from $[c, c-1]$;

If $L1$ is parallel to $L2$ and $L3$, then $L1$ and $L2$ or $L1$ and $L3$ are either identical or they do not intersect. In the former case $\gamma(p)$ is equal to $polygon[a-1]$. In the latter case $\gamma(p)$ does not exist.

Conversely, if $L1$ is not parallel to $L2$ and $L3$, then two intersection points exist, P1= $L1 \cap L2$ and P2 = $L1 \cap L3$. As stated in Klee and Laskowski (1985) only the points which are on the same side of the line $[c, c-1]$ as the polygon will be considered. Therefore, $\gamma(p)$ will be equal to the intersection point which is on the same side of the line $[c, c-1]$ as the point $polygon[c+1]$.

Algorithm 21 Check if gamma of the point exists and compute it when possible

```
1: function GAMMA(index, gammaPoint)
2:    if (!FindGammaIntersectionPoints(index, polygon[a], polygon[a − 1],
3:       polygon[c], polygon[c − 1], intersectionPoint1, intersectionPoint2)) then
4:       return false;
5:    end if
6:
7:    if intersectionPoint1 and polygon[c + 1] are on the same side
8:       of line [c, c − 1] then
9:       gammaPoint ← intersectionPoint1;
10:   else
11:      gammaPoint ← intersectionPoint2;
12:   end if
13:
14:   return true;
15: end function
```

Finding $vertexC$ on side B is similar to computing gamma of a point.

Algorithm 22 Find $vertexC$ on side B with the property that $h(vertexC) = 2 \times h(a-1)$

```
1: function FINDVERTEXCONSIDEB
2:    if (!FindGammaIntersectionPoints(polygon[a − 1], B.StartVertex,
3:       B.EndVertex, C.StartVertex, C.EndVertex, intersectionPoint1,
4:       intersectionPoint2)) then
5:       Write "There is an error in the implementation of the algorithm!";
6:    end if
7:
8:    if intersectionPoint1 and polygon[c + 1] are on the same side
9:       of line [c, c − 1] then
10:      return intersectionPoint1;
11:   else
12:      return intersectionPoint2;
13:   end if
14:
15:   return true;
16: end function
```

Algorithm 23 Find intersection points for determining gamma

1: **function** FINDGAMMAINTERSECTIONPOINTS($index$, $line1StartVertex$, $line1EndVertex$, $line2StartVertex$, $line2EndVertex$, $intersectionPoint1$, $intersectionPoint2$)
2: $L1 \leftarrow$ line determined by $line1StartVertex$ and $line1EndVertex$;
3: $L2$, $L3 \leftarrow$ lines parallel to and at a distance of (2 x $h(index)$) from the
4: line determined by $line2StartVertex$, $line2EndVertex$;
5:
6: **if** $L1$, $L2$, $L3$ are parallel **then**
7: **if** ($L1$ identical to $L2$) OR ($L1$ identical to $L3$) **then**
8: $intersectionPoint1 \leftarrow line1StartVertex$;
9: $intersectionPoint2 \leftarrow line1EndVertex$;
10: **else**
11: **return** false;
12: **end if**
13: **else**
14: $intersectionPoint1 \leftarrow$ intersection of L1 and L2;
15: $intersectionPoint2 \leftarrow$ intersection of L1 and L3;
16: **end if**
17:
18: **return** true;
19: **end function**

5 Results

The overall complexity of the algorithm is $\Theta(n)$ where n represents the number of vertices defining the convex polygon.

A benchmark was set up to check the linearity of a C++ implementation of the algorithm. The variable of interest n represents the number of points defining the convex polygon. The considered values for n were chosen from the range 100 to 10,000 with a step size of 100.

For each value of n 100 random convex n-gons were generated using the Computational Geometry Algorithms Library (CGAL) (CGAL 2013). Thus, the total number of convex polygons in the benchmark is 10,000. Illustrative randomly generated 100-gons and their corresponding minimal area enclosing triangles are depicted in Fig. 4.

The advantage of creating such a data set is that it could potentially be employed for testing the efficiency of other similar applications; see http://people.brunel.ac.uk/~cspgoop/data/notes/2014/min_enclosing_triangle for more details.

The execution time was measured in microseconds (μs). Since the algorithm is linear the execution times were relatively short. Execution times differences at small scales (e.g.

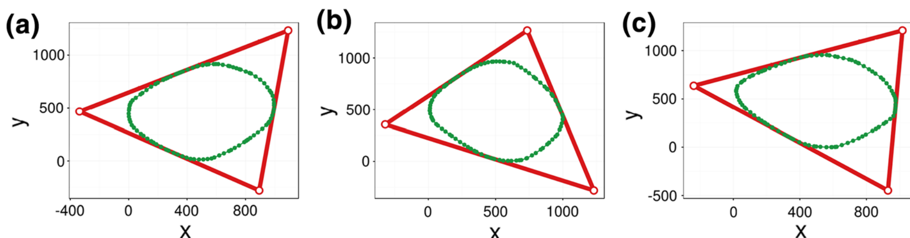

Fig. 4 Randomly generated 100-gons and their corresponding minimal area *enclosing triangles*.*Polygons* are depicted in *green* and the *enclosing triangles* in *red*. Coordinates of the 2D points defining the polygons and their enclosing triangles are provided in Online Resource 1

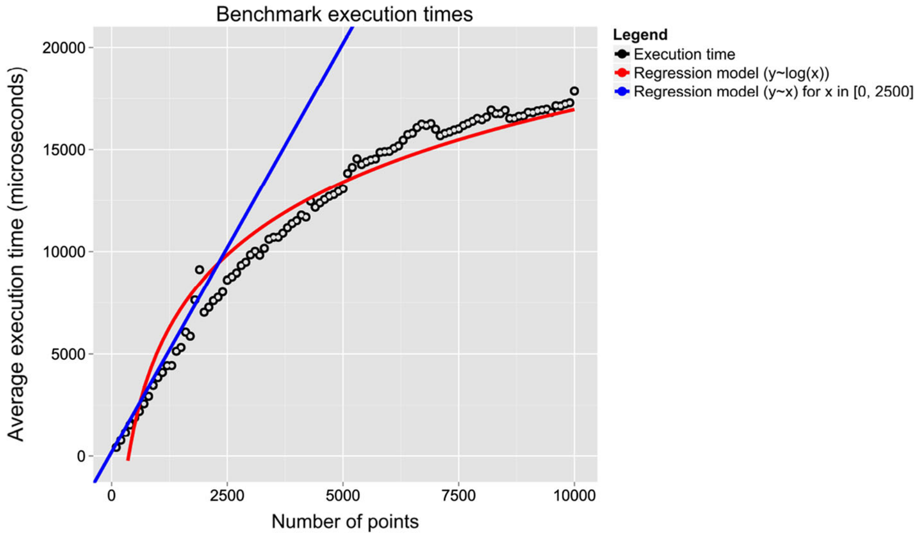

Fig. 5 Average results of 100 executions of the benchmark (i.e. 100 × 10,000 executions). *Black points* represent the mean of the execution times for a fixed value of n (number of points defining the convex polygon). The *red* and *blue lines* are the lines fitted to the obtained set of execution times using the formulae $y \sim \log(x)$, respectively $y \sim x$

microseconds) can be influenced among others by processes running in the background and the operating system. In order to overcome this issue the benchmark was run 100 times and only the mean of the execution times for each distinct value of n was considered.

All tests have been performed on a regular computer (Intel(R) Core(TM) i7-4700MQ CPU @ 2.40 GHz, 16.0 GB DDR3 RAM, Windows 8 x64) in a sequential manner. The obtained results are depicted in Fig. 5.

The black points represent the mean of the execution times for a fixed value of n; the red and blue lines are the lines fitted to the obtained set of execution times using the formulae $y \sim \log(x)$, respectively $y \sim x$. According to the obtained results the execution time increases linearly for polygons defined by approximately 0–2,500 points, respectively logarithmically for polygons defined by approximately 2,500–10,000 points, with respect to n. In brief the implementation of the algorithm scales well with respect to the value of n. However, considering that the complexity of the algorithm is $\Theta(n)$ our expectation was that the algorithm implementation would scale linearly, and not logarithmically, with respect to n. Explaining the cause of this behaviour could be a potentially interesting research question, which is however not pursued here as it goes beyond the scope of this note.

6 Correctness

The executions described in Sect. 5 verified empirically that the algorithm implementation scales (sub-)linearly with respect to n. However, they did not assess the correctness of the computed minimal area enclosing triangle. In order to address this challenge three verification approaches were employed. Let us denote the expected optimal minimal area enclosing triangle by OT, and the minimal enclosing triangle computed by the linear algorithm by CT.

The first verification approach relies on generating regular convex n-gons, $n = 3k$ (in our case $k = \overline{1, 3334}$), for which the minimal enclosing triangle is known to be equilateral. Given the coordinates of the 2D points defining the n-gon, OT can be automatically computed in $O(1)$. If the minimal enclosing triangle CT computed by the algorithm implementation matches OT the algorithm implementation is considered valid. Otherwise it is invalid.

The second verification approach builds on the first one by applying affine transformations AT to each generated regular convex n-gon, and thus obtaining a new convex polygon; in our case $AT = \{$scaling by a factor of 1.5 with respect to both Ox and Oy, counterclockwise rotation by $\pi/4\}$. According to Klee and Laskowski (1985) the optimal enclosing triangle OT of a transformed polygon can be determined by applying the same affine transformations AT to the optimal enclosing triangle computed for the initial non-transformed regular polygon. Similarly, to the first verification approach OT can be determined in $O(1)$. Moreover, the algorithm implementation is validated by checking if OT matches CT.

The main advantage of the first two verification approaches is that the optimal minimal enclosing triangle OT can be determined in $O(1)$. Conversely their main disadvantage is that only polygons with specific properties are considered, while the linear algorithm is general purpose and should work for any convex polygon.

In order to address this limitation the third verification approach checks if the results of a brute-force algorithm (computing minimal enclosing triangles in $O(n^3)$, checking their validity in $O(n)$), which is guaranteed to check all possible minimal enclosing triangles, matches the results of the linear algorithm described in this note. The main advantage of this approach is that any convex polygon can be considered. Conversely its main disadvantage is that the brute-force algorithm implementation does not scale well with n. Therefore the value of n was limited to the range [3, 200] during our tests. Moreover, only ten polygons were randomly generated for each value of n.

All verification approaches described above have been implemented in C++ and were employed to validate the linear minimal area enclosing algorithm implementation; the execution of all tests ended successfully empirically confirming that the algorithm implementation is valid. For reproducibility purposes the implementation of all algorithms and verification approaches (including generated datasets) are made freely available at https://github.com/IceRage/minimal-area-triangle.

7 Conclusions

The steps which were not described in the original paper (O'Rourke et al. 1986) have been presented in detail here such that the implementation is easy to reproduce. A step by step execution of a simple example was described in Sect. 3 in order to illustrate how the minimum enclosing triangles are found. The results of the benchmark execution indicate that the algorithm is linear and scales well with respect to the number of points defining the convex polygon. Moreover, three different verification approaches were used to assess the correctness of the algorithm implementation.

The algorithm was implemented and tested in C++. A version of the algorithm was implemented and added to the *imgproc* module of the OpenCV library. It takes a 2D point set as input and computes its convex hull before finding the minimum enclosing triangle; see https://github.com/Itseez/opencv/blob/master/modules/imgproc/src/min_enclosing_triangle.cpp for more details. A usage example, documentation of the new functionality and unit tests have been added to the OpenCV library as well.

Finally, errors might occur when comparing real numbers due to the finite floating point precision of computers. Depending on the configuration of the system, the compiler and the representation of 2D points a different tolerance value for comparing real numbers should be used; see Dawson (2012) and Goldberg (1991) for more details.

Acknowledgments The authors gratefully acknowledge the insightful comments provided by Joseph O'Rourke which helped improve the quality of the manuscript. Ovidiu Pârvu is supported by a scholarship from Brunel University.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

Bradski G, Kaehler A (2008) Learning OpenCV. In: Computer vision with the OpenCV library. O'Reilly, New York

CGAL (2013) Computational geometry algorithms library. http://www.cgal.org

Chandran Mount DM (1992) A parallel algorithm for enclosed and enclosing triangles. Int J Comput Geom Appl 2(2):191–214. doi:10.1142/S0218195992000123

Dawson B (2012) Comparing floating point numbers, 2012 edn. http://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Goldberg D (1991) What every computer scientist should know about floating point arithmetic. ACM Comput Surv 23(1):5–48

Klee V, Laskowski MC (1985) Finding the smallest triangles containing a given convex polygon. J Algorithms 6(3):359–375. doi:10.1016/0196-6774(85)90005-7. http://www.sciencedirect.com/science/article/pii/0196677485900057

O'Rourke J, Aggarwal A, Maddila S, Baldwin M (1986) An optimal algorithm for finding minimal enclosing triangles. J Algorithms 7(2):258–269. doi:10.1016/0196-6774(86)90007-6. http://www.sciencedirect.com/science/article/pii/0196677486900076

Toussaint GT (1983) Solving geometric problems with the rotating calipers. In: Proceedings of the IEEE Melecon, vol 83, p A10. http://web.cs.swarthmore.edu/adanner/cs97/s08/pdf/calipers.pdf