Solution of the test

Grishchenko Dmitry

Problem 1

In this problem, we consider Delaunay triangulation algorithm described in [1][Chapter 9] for the specific set of points $P = \{p_1, \ldots, p_n\}$. where each p_i is a 2-dimensional point with **Int32** coordinates.

To answer on the proposed question, we need to understand which lines of the algorithm requires for "longer" datatype.

First, in the algorithm we find two additional points p_{-1} and p_{-2} that could be larger than Int32; more precisely, if P consists of all 4 limit points of Int32 for p_{-1} we will need to work with 3×INT_MIN that requires for Int64 datatype.

Another 2 operations that are performed by the algorithm are :the procedure of legalizing the edge and the test whether the given point is inside or on the border of the triangle.

The only hard part of the legalizing procedure is to check if the point is inside of the circumcircle of the Int32 triangle. Let us first provide an algorithm for this check and then come up with a conclusion on the "size" of the long int required for it.

Since we are working with integer types only, the circumcircle could not be interpreted as a pair of its center and radius for this validation test. However, there is an integer way to check if the point inside of the circle. Let us consider triangle (p_1, p_2, p_3) and x_i is a point to check (see Figure 1). (As we could see from the figure, points y_i are outside of the circle.)

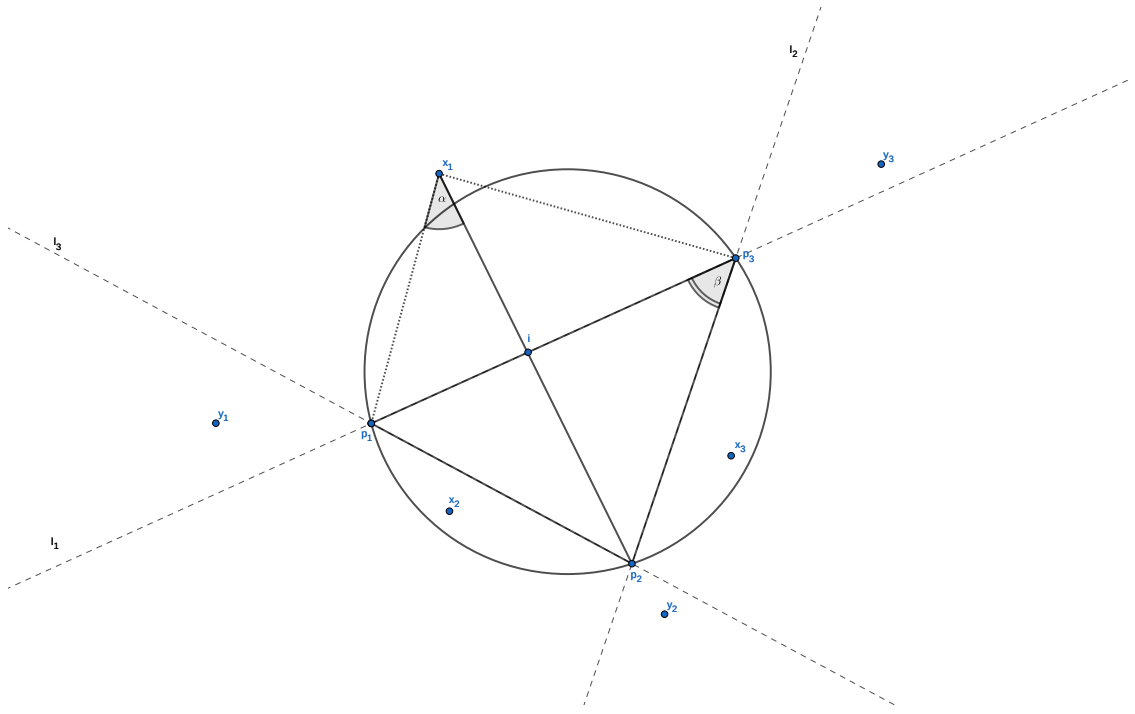


Figure 1

First, let us check if the point is inside of the triangle. If point x is in the triangle (p_1, p_2, p_3) then the

sum of areas of $(p_1, p_2, x), (p_1, p_3, x)$, and (p_2, p_3, x) is equal to the area of (p_1, p_2, p_3); moreover, if one of the areas is equal to 0 the point is on the corresponding edge of the triangle. So this test requires Int96 if the outer product is used to compute the area (an absolute value of (1)).

If the point is outside of the triangle let us check in which "sector" of the plane it is; in other words we need to understand if the point of the type y or of the type x and furthermore to specify the subscript. To do it, for every line l_i let us understand if the point is "above" the line or not. For this, let us calculate the oriented area of triangle (p_1, x_2, p_3) and if it is positive, then the point x_2 is above the line (if it is 0 it is on the line and if it is negative it is under the line). By the definition the doubled signed area is

$$2S = (x^2 - x^1)(y^3 - y^2) - (x^3 - x^2)(y^2 - y^2). \tag{1}$$

where $p_1 = (x^1, y^1), p_2 = (x^2, y^2)$, and $p_3 = (x^3, y^3)$. It is easy to see, that $2S$ could be calculated using Int128. In the very special case when the point is on the line an easy check if the point is between the ends of the segment would answer to the question if the point is inside of the circle.

Without loss of generality let us consider that our point is x_1. From the school geometry classes it is known, that for point x_1 the necessary and sufficient condition to be inside of the circle (or on the border) is : angle α is greater or equal than β. This corresponds to $\cos(\beta) \geq \cos(\alpha)$.

Since the division procedure does not preserve integers to compare two cosines we should compare the following terms (the sign $\leq$ implies that point x_1 is inside of the circle.)

$$\langle \overline{x_1p_1}, \overline{x_1p_3} \rangle \|\overline{p_2p_1}\|_2 \|\overline{p_2p_3}\|_2 \quad \text{Vs} \quad \langle \overline{p_2p_1}, \overline{p_2p_3} \rangle \|\overline{x_1p_1}\|_2 \|\overline{x_1p_3}\|_2.$$

However, since the distance is not integer we should operate with a squares of both sides

$$\underbrace{(\langle \overline{x_1p_1}, \overline{x_1p_3} \rangle)^2}_{\text{Int128 + small amount} \leq 6} \qquad \underbrace{\|\overline{p_2p_1}\|_2^2}_{\text{Int64 + small amount} \leq 3} \qquad \|\overline{p_2p_3}\|_2^2 \quad \text{Vs} \quad (\langle \overline{p_2p_1}, \overline{p_2p_3} \rangle)^2 \|\overline{x_1p_1}\|_2^2 \|\overline{x_1p_3}\|_2^2.$$

that requires for **Int512** (or **Int268** if the specific types are allowed).[1]

Better solution.

Let us recall the following theorem.

Theorem 1. *If a, b, c, and d form a convex polygon then d lies in the circle determined by a, b, and c iff :*

$$\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & a_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix} > 0 \tag{2}$$

Since we know the position of point we could work with convex polygon (e.g., for x_1 the polygon would be $a = p_1, b = p_2, c = p_3$, and $d = x_1$). Now, we could see that it is enough to calculate the sign of the determinant. Unfortunately for this we need to calculate determinant itself and for this we need to figure with objects of type $a_x * b_y * (c_x^2 + c_y^2)$ that requires **Int192**.

Another solution

Finally, we could calculate the intersection of diagonals i of the polygon x_1, p_1, p_2, p_3 and check if

$$\|\overline{x_1i}\|_2 \|\overline{p_2i}\|_2 > \|\overline{p_1i}\|_2 \|\overline{p_3i}\|_2$$

that will correspond to the outer position of point x_1. However the intersection of two lines is rational with enumerator and denominator both being bigger than Int64. That requires scaling of ALL points to have

[1] Coordinates of $\overline{x_1p_1}$ (the same is true for all the other vectors) could be twice bigger than the limits of Int32 so for the scalar product we should use Int64 + 3 extra bits.

integers that leads to the bigger than Int128 for the intersection and bigger than Int96 for the vertices. Finally, to operate with integers we should consider squares of the distances instead of the distances itself that leads to the **Int512** at least that is worse than in previous case so we won't provide a precise analysis here..

Answer.

In conclusion, the most "size-consuming" operation requires **Int192**.

Problem 2

In this problem, we create class *cRegularPolygon* (class tOctet in the code) that corresponds to the convex polygon with regular 45 edges. The object of this class should support all the provided public methods. Moreover, we provide some unit-tests to show that all the methods works correct. (We consider that all the points are integers and all the intersections of edges are integers as well so we do not need to provide the tools for odd-point analysis)

Structure of the class. Every object of the class could be represented by its edges; however this is not useful for the required methods. To make it more practical, we propose representing the object by *limits* that are more operation-tolerant. More precisely, every regular direction line corresponds to the equation $ax + by = c$, where a and b depend only on the direction, however c corresponds to the *limit*. Furthermore, we consider the limit of cRegularPolygon in direction *eDir* d to be a limit of edge that has $d + 2$ direction while moving in anticlockwise direction.

Using this structure any regular polygon could be represented (starting from point (0 edges) up to the full octagon (8 edges)).

Constructors. To make the class useful we should equip it with different constructors. The default one that creates an object with all limits equal to 0. Second type of constructor is by point (class tPoint) that is a pair of 2 Int32 coordinates (x, y). This constructor should calculate the limits of the point and fill the attribute of the class. Third, we consider constructor by the pair of points (class tSegment that is an ordered pair of 2 tPoints.) We also provide a constructor by any list of points that creates octet by point and then uses Combine (to be described after) procedure to add all the points from the list. More precisely, this constructor creates the minimal octet that contains all the points. Finally, we provide the copy constructor. Since we are not allocating the memory by *new* we do not need different from default destructor.

Auxiliary classes. As we mentioned above, class tOctet considers two auxiliary classes tPoint and tSegment. As we already said, we would need to calculate the limits of these objects to construct tOctet so we provide this functionality to these classes. Moreover, we add basic operators to operate with points like with vectors ($'+', '-'$ and scalar multiplication). Also we add functionality of the shift (add tPoint) to the tSegment object. Furthermore, we need to operate with tSegments objects in future so we provide *direction* attribute that is a single (the pair of them) for the regular segment and a pair of neighbour directions for the odd-angle one.

Realization of functions. For all functions for the simplicity we assume that all the points has small coordinates so the arithmetical operations such as sum and difference of two points returns a point as well. It is quite practical assumption and it makes all algorithms easier. We understand, that some additional functionality should be added to work with real Int32 points; however for this test we assume it is unnecessary.

Moreover, we consider all the diagonals to intersect in integer points to simplify the work and get rid of oddpoint type.

Cover point. The action of this function is quite easy, we just make all the limits of octet that are smaller then the corresponding limits of the point to be the same as for the point.

To check that point **Is inside** of the octet we check if all it's limits are not greater than the corresponding limits for the octet. If one (or some) of them are equal to the octet's one than the point is on the edge (is vertex).

Common point. Function common point works for 2 neighbour directions for now. Using the switch-case construction it uses an explicit formula for the every pair of the neighbour directions.

Has edge. In this function we check that the limit is smaller than the sum(for odd direction) or average (for orthogonal direction) of two neighbour directions.

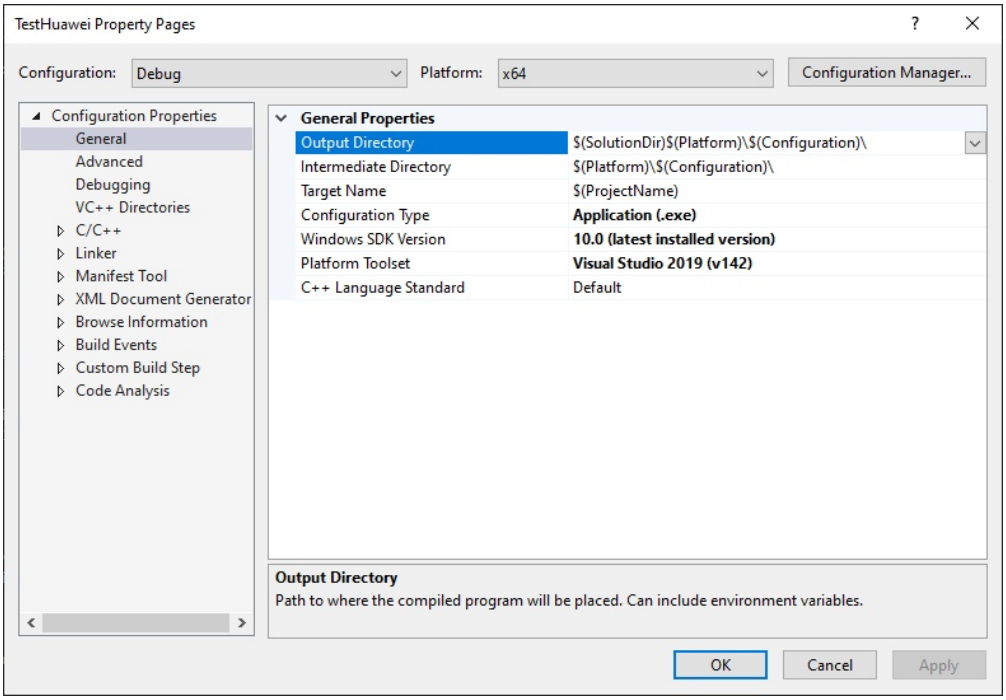
Intersect. This function returns *std::shared_ptr<tOctet>* to avoid memory leaks. The intersection if it exists is an octet with every limit be the minimal of the corresponding limits of these two octets. However, the intersection is not empty if for all directions $oct1.Limit(dir) + oct2.Limit(dir+4) \geq 0$.

Cure. Is an auxiliary function that postproceed the intersection procedure to avoid stripes. It assumes that octet is non-empty and consists of 2 steps. First is to avoid big diagonal limits that is for all diagonal limit we make it equal to the sum of the neighbour limits if it is bigger. Second, for all orthogonal limits (assume E) we calculate the limit of orthogonal directions (N and S) in 2 different ways: using Limit(S)/Limit(N) for the octet and using (+-)(Limit(NE)-Limit(SE))/2. In case when the second value is bigger we make the corresponding diagonal and basic orthogonal (E) limits smaller to fit.

Inflate. This easy procedure is just add the required amount for limits (inflate parameter for orthogonal) and the scaled by $\sqrt{2}$ for diagonal. However, there are two points. First, if inflate parameter is negative we do inflation if and only if the octet is inflatable that is if we proceed the procedure the octet should not become empty. Second, if we perform inflate with parameter d and then with parameter $-d$ the result should be the same as before inflation that calls for the correct rounding in case of negative inflation.

General remarks.

The project is created in MS VS 2019 and contains boost library for unit-tests. The library is a part of the project and for the correctness of the compilation the following general properties should be used.



Problem 3

First, let us provide the general idea about the way we want to solve this problem. If the segment is oddangle (irregular) then there are two neighbour directions such that the direction of the segment is in between them. Moreover, let us consider tOctet that is the minimal one that contains this two points. Then it has edges only in the directions orthogonal to these neighbour ones and in addition the edges are the lower/upper two segments regularization of the segment in case when there is no obstacles.

The second idea is that any feasible regularization of the shortest length consists of the segments of these two directions only so it forms *ladder*.

Combining these two ideas we perform the following algorithm to regularize AB.

Step 1 Find two neighbour directions dir1, dir2 such that dir2 = Next(dir(1));

Step 2 create shadows from all obstacles;

Step 3a go up to the shadow with dir1;

Step 3b go up to the segment AB;

Step 3c go to Step 3a if we are not in B.

To make it clear that algorithm performs the required result let us clarify what is the create shadows procedure and why it is too important here.

For the simplicity, let us consider that dir1 = SE and dir2 = E. For any point X that is a current point of our regularization we can move in the direction dir1 up to the maximal of the octet limits in (dir2+2) where the maximum is selected between all the octets that has bigger Limit(dir1+2) than point X (See Figure 2) sine otherwise we can not move with required directions without the intersection. (Since we are not allowed to jump through the obstacles).

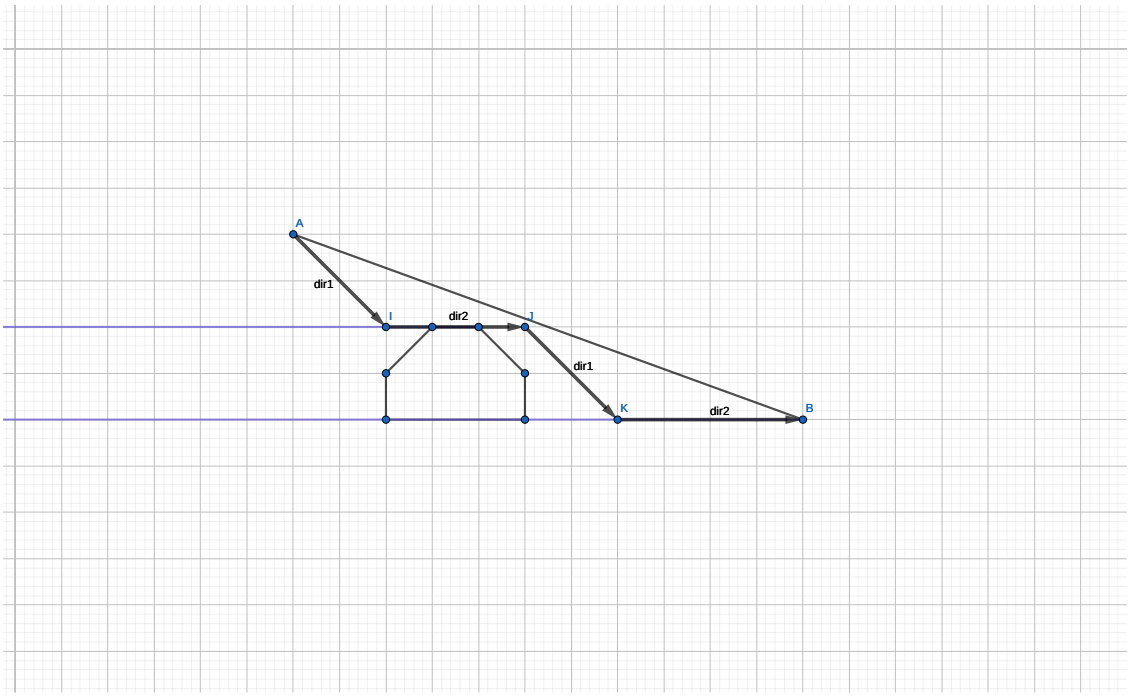


Figure 2

As we see, the real constraint for us is a set of rays that goes from the common vertex for (dir1+2, dir2+2) in direction -dir2. In general, when the obstacles are possible from the both sides and we are not restricted to be below the segment AB the same story with shadows takes place while we going along dir2. However, we have another restriction wile going along dir2 - we should not intersect the segment AB and furthermore, we should stop in the integer point.

This could be realized in two different ways. First, using binary search for the segment with start point I and end point having the same Limit(dir2) as B.

Second way is about to find an intersection. More precisely, let us denote by $l_1(l_2)$ the difference in limits Limit(dir1) (Limit(dir2)) for points A and B. Let us also denote by l'_1 the difference between the current point (see point I in the figure) and point A. Then the difference for the next point (see point J in the figure) is static_cast<int>(std::floor($l_2/l_1 * l'_1$)) that will give exactly the point we needed since the dir1 and real direction of segment AB are not so close due to the fact that point I and J are different for solution to exist.

It is easy to see that this algorithm provides the line with the smallest amount of segments among all that correspond to the constrains.

Implementation details.

The only thing to shed the light on is the structure of the shadows to make the process fast.

The class is initialized by the segment AB. We propose to store all the shadows in the map with key being a limit in dir2 and value is a current "height of the ladder". We should provide the class with functions to add an obstacle that is equipped with semantic check to not add an obstacles that above (that are added inn the top shadow in general). To make this check (under the assumption that there is no violation) it is enough to check if any of the vertices of obstacle is above the segment. Before this we want to skip all the obstacles that are far from the segment; more precisely we intersect all obstacles with the octet generated by segment and if the intersection is empty or regular segment we skip this obstacle.

Let us assume that we check the top-right point of the obstacle in our picture. Using the same notation as in the previous paragraph (define l'_2 the same way as l'_1) we understand that if $l_2 l'_1 \leq l_1 l'_2$ then the point is under.

Problem 4

This problem is quite similar to the previous one. Let us show it. First, we have an additional parameter of every segment - width that requires to keep some distance to the obstacles. However, instead of this, we could inflate all obstacles on this parameter and keep going with 0 width segments.

Another difference is several segments mode. Since the segments are non-regular an octet does not correspond to the area with distance smaller than required. To figure it out we should consider the sequence of segments such that all the next segment is above all the previous. In these case, all the non-regular segments are not obstacles for the algorithm. However, all the processed segments now are the regular trajectories (the consequence of regular segments) so the obstacle area could be well-approximated by the union of the inflated octets corresponding to every segment with inflation parameter $(w_1 + w_2)/2$. It means, that for every next segment the problem 4 reduces to problem 3.

As a result, the only things to implement are the procedure that orders the segments and the one that provide the list of obstacles by the regular trajectory and inflate parameter.

The second is quite trivial and the first one requires some idea. Let us consider two non-intersecting segments AB and CD. Then, at least one of the following properties holds: both C and D are from the same side of line that contains AB or both A and B are from the same side of line that contains CD (see solution of problem 1 to see how to do it). It allows to order all the segments in the plane (see e.g., Figure 3). Let us proof it using the principle of the mathematical induction.

Base: for any two segments we can order them.

Inductive step: let us assume that for $n-1$ segments we can order.

Let us add n-th segment. Let us compare it with the minimal of the $n-1$ is it is smaller than it than it is smallest one; otherwise, the smallest one is the same as for $n-1$ and we should order all the others that

is possible due to the assumption.

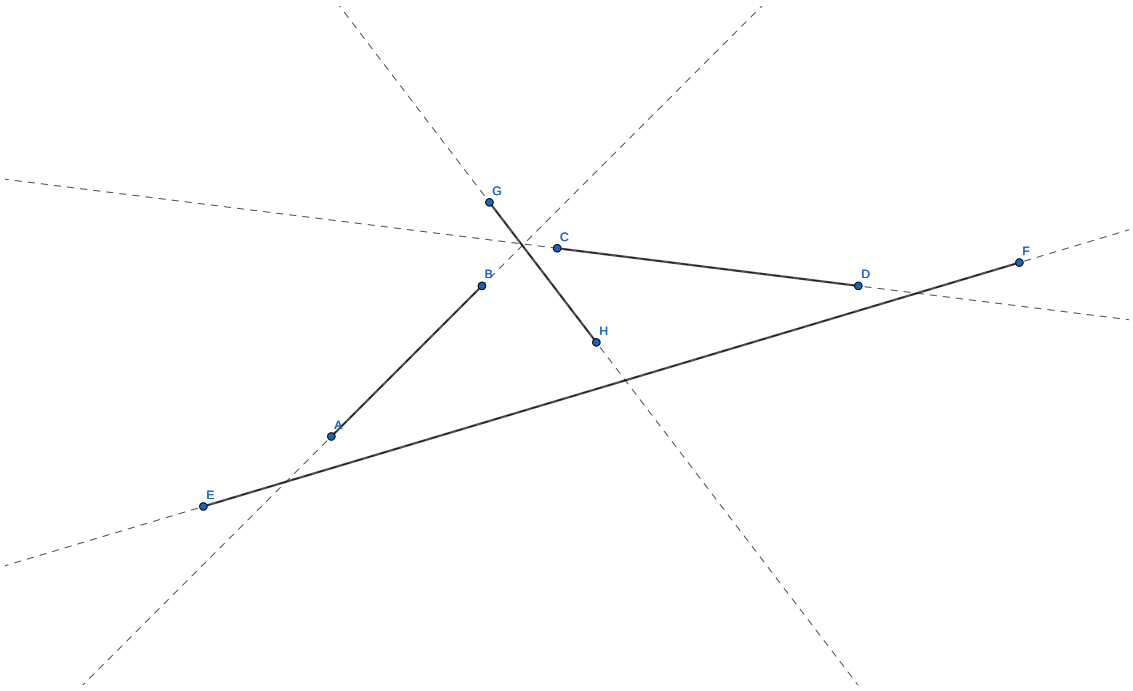


Figure 3: The order of the segments is the following $CD > GH > AB > EF$

Testing

To test all the functionality we use boost unit tests with BOOST_CHECK.

References

[1] De Berg, M., Cheong, O., Van Kreveld, M., Overmars, M.: Computational geometry: Algorithms and applications. santa clara (2008)