

Another Classroom Example of Robustness Problems in Planar Convex Hull Computation

Marc Mörig

Department of Simulation and Graphics, Faculty of Computer Science,
University of Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany
`moerig@isg.cs.uni-magdeburg.de`

Abstract. Algorithms in computational geometry are designed under the assumption of exact real arithmetic. Indiscriminately replacing exact real arithmetic by hardware floating-point arithmetic almost inevitably leads to robustness problems. Kettner et al. provide examples where rounding errors let such straightforward implementations of incremental convex hull computation crash, loop forever, or silently compute garbage. We complement there work by providing problematic examples for another planar convex hull algorithm.

Keywords: implementation, numerical robustness problems, floating-point geometry

1 Introduction

Algorithms in computational geometry are designed under the assumption of exact real arithmetic at unit cost [5]. Simply using hardware floating-point arithmetic as an indiscriminate substitute in implementations almost inevitably leads to robustness problems [6,8]. However, there are only few examples documented in the literature: The LEDA book [4] gives some instructive examples in section 9.6 and reports on experiments that fail for floating-point based implementations in sections 10.7 and 10.8. Shewchuk [7] discusses an example where the computation of a 2D Delaunay triangulation by divide and conquer fails. Moreover, it is folklore that straightforward implementations of Jarvis' march for computing planar convex hulls can loop forever for certain nearly degenerate input data. Computing the convex hull of a set of points in the plane is one of the best studied problems in computational geometry. Kettner et al. [3] show how to create input data that lets an implementation of incremental planar convex hull computation fail in various ways. After fixing an implementation, they examine the decisions the implementation will make for input points in a certain critical range of space. This allows them to select input points where these decisions are incorrect and lead to failures. Failures generated include not only computing output that is not valid, but also infinite looping, or the program crashing. However, the planar convex hull algorithm they study does not have worst-case optimal running time. The same holds for Jarvis' march. Both have quadratic worst-case complexity.

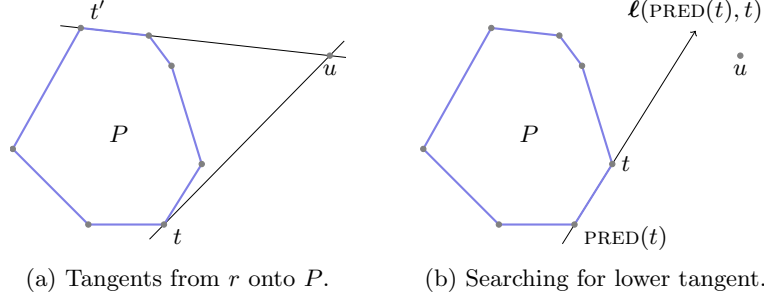


Fig. 1. Basic steps in a 2D convex hull computation.

We complement the work by Kettner et al. by providing examples for the plane sweep variant of Graham scan [1], a convex hull algorithm with optimal worst-case optimal running time $O(n \log n)$.

2 Short Description of Algorithm and Predicates

Let p , q , and r be three points in the plane, and let $\ell(p, q)$ be the oriented line passing first through p and then through q . The 2D orientation predicate determines the position of r relative to $\ell(p, q)$. If $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$, then the predicate is tantamount to computing the sign of the determinant

$$D_{O2} = \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - p_x & r_y - p_y \end{vmatrix}. \quad (1)$$

The three points p, q and r are collinear if and only if D_{O2} is zero. Otherwise, r is to the left of $\ell(p, q)$, if D_{O2} is greater than zero and r is to the right of $\ell(p, q)$, if D_{O2} is smaller than zero. Besides coordinate comparison this is the only predicate used in the algorithm.

The algorithm itself is a variant of Grahams scan [1] and proceeds as follows. We process the points one by one, in xy -lexicographical order. We maintain the convex hull P of already processed points as the circular sequence of vertices in counterclockwise order along its boundary. Thus, when we arrive at a new point u , we have to update P . Since we process points from left to right, u is not contained in P . Consider the two tangents from u onto P , see Figure 1a. Each tangent touches exactly one or two vertices of P . Let t' be the vertex of P furthest from u that the upper tangent touches, and let t be the vertex of P furthest from u that the lower tangent touches. Note that $t = t'$ is possible in case P is a segment. In the sequence of vertices of P , we replace all vertices between t and t' with u .

How do we find t and t' ? Let q be the point that was processed in the previous update step. Then q is the rightmost vertex of P . We start with $t = q$ and check

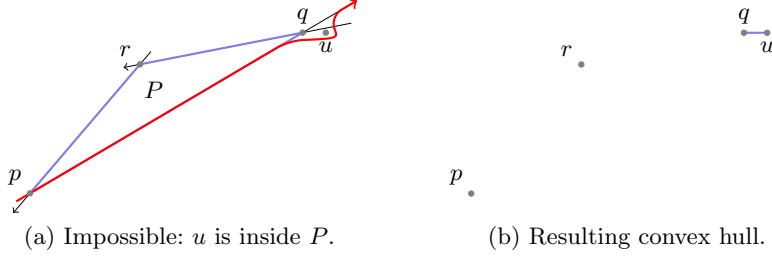


Fig. 2. A single incorrect orientation result can lead to catastrophic failure.

the position of u relative to $\ell(\text{PRED}(t), t)$ with the 2D orientation predicate, see Figure 1. As long as u is not on the left side, we advance t to $\text{PRED}(t)$ and check the position of u again. The vertex t' can be found analogously. There is one exceptional case: if P is a segment we have to stop the search for t and t' after at most one step.

As for the 2D orientation predicate we use the straightforward implementation. We compute an approximation D'_{O_2} of D_{O_2} as

$$D'_{O_2} = (q_x \ominus p_x) \otimes (r_y \ominus p_y) \ominus (r_x \ominus p_x) \otimes (q_y \ominus p_y), \quad (2)$$

where $\otimes$ and $\ominus$ are floating-point multiplication and subtraction. The point p is distinguished in the computation of D'_{O_2} and is called the pivot point. Many familiar mathematical properties like associativity or distributivity do not hold for floating-point operations. Therefore, while permuting p , q , and r will lead to sign changes only in D_{O_2} , this is in general not the case for D'_{O_2} . Our results can be reproduced using floating point arithmetic compliant to the IEEE 754 standard [2].

3 How It Fails

What can go wrong with this algorithm when a floating-point based orientation predicate is used? The search for t and t' may stop too soon, in which case the resulting polygon is not convex anymore. This may lead to incorrect output or more problems in later steps, since the correctness of the search for t and t' depends on P being convex! The search may also stop too late, in that case vertices of P are cut away and may not be contained in the end result.

Carrying the first case to the extreme, the search for both t' and t may stop at q . This can never occur geometrically, since for full dimensional P the upper and lower tangent touch P in different vertices. But it can occur due to a single incorrect result from the 2D orientation predicate, as illustrated in Figure 2a. Here the current hull polygon P has vertices p , q and r and is about to be updated with point u . Figure 3 shows the lattice of points with floating-point coordinates close to q , among them u , and how the 2D orientation predicate classifies their

Let

$$\begin{aligned}
 p &= (-10.04094770362331879, -7.506293383338360492) \\
 q &= (1.056089924324703055, -0.9655180522057801307) \\
 r &= (-5.5608992432470305545, -2.255180522057801307) \\
 u &= (1.0560899243247048318, -0.9655180522057800196) \\
 u' &= (1.0560899243247043877, -0.9655180522057800196)
 \end{aligned}$$

In the drawings below, we classify each point with floating-point coordinates near q if it is to the right ($-$), to the left ($+$), or on ($\bullet$) line ℓ drawn solid. Correctly classified points are drawn in grey, while misclassified points are colored ($-$, $+$, $\bullet$).

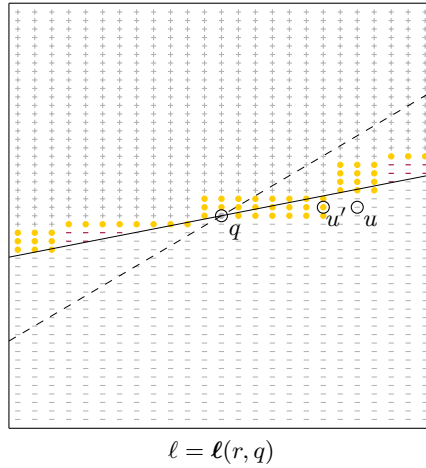
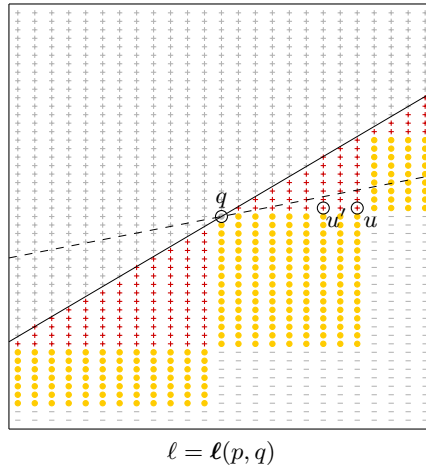


Fig. 3. Zoom in on points q and u in Figure 2.

position relative to $\ell(p, q)$ and $\ell(r, q)$. Point u is correctly classified to be right of $\ell(r, q)$, but incorrectly classified to be left of $\ell(p, q)$. Since we process points in sorted order, u must be outside P , but the incorrect classification moves u inside P , at least from the view of the algorithm. We have $t = t' = q$ and all vertices between t and t' are to be replaced by u . The implementation may now traverse the sequence of hull vertices, starting at t , and remove vertices until t' is reached. This will remove all vertices except q . Then u is inserted, resulting in the hull polygon shown in Figure 2b.

For another example, suppose we are updating the current hull P with the point u' , two positions to the left from u in the lattice of floating-point points in Figure 3. The point u' is incorrectly classified to be on $\ell(r, q)$, and incorrectly classified to be left of $\ell(p, q)$. Hence, u' will be inserted into the sequence of vertices between q and r , resulting in a non-simple hull polygon.

4 Conclusions

Computing the convex hull of a set of points in the plane is a problem studied in almost all introductory courses in computational geometry and hence a prime candidate for illustrating robustness problems. We provide examples of failure for worst-case optimal planar convex hull computation, complementing the work of Kettner et al.

In general however, floating-point arithmetic guarantees small errors and computes quite accurately. This observation can be quantified and exploited for efficient exact geometric computation [6,8].

References

1. de Berg, M., Cheong, O., van Krefeld, M., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer, 3rd rev. edn. (2008)
2. ANSI/IEEE Standard 754-1985: IEEE Standard for Binary Floating-Point Arithmetic (1985), reprinted in SIGPLAN Notices, 22(2):9-25, 1987
3. Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., Yap, C.K.: Classroom examples of robustness problems in geometric computation. Computational Geometry: Theory and Applications 40(1), 61–78 (May 2008)
4. Mehlhorn, K., Näher, S.: LEDA: A platform for combinatorial and geometric computing. Cambridge University Press, Cambridge (November 1999)
5. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer, 1st edn. (1985)
6. Schirra, S.: Robustness and precision issues in geometric computation. In: Sack, J.R., Urrutia, J. (eds.) Handbook of Computational Geometry, chap. 14, pp. 597–632. Elsevier, Amsterdam, The Netherlands (January 2000)
7. Shewchuk, J.R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete and Computational Geometry 18(3), 305–363 (1997)
8. Yap, C.K.: Robust geometric computation. In: Handbook of Discrete and Computational Geometry, chap. 41, pp. 927–952. CRC, 2nd edn. (2004)