Classroom Examples of Robustness Problems in Geometric Computations*

Lutz Kettner† Kurt Mehlhorn† Sylvain Pion‡ Stefan Schirra§ Chee Yap¶

April 2, 2004

Abstract

The algorithms of computational geometry are designed for a machine model with exact real arithmetic. Substituting floating point arithmetic for the assumed real arithmetic may cause implementations to fail. Although this is well known, there is no comprehensive documentation of what can go wrong and why. In this paper, we study simple algorithms for planar convex hulls and 3d Delaunay triangulations and give examples which make the algorithms fail in many different ways. For the incremental planar convex hull algorithm our examples cover the negation space of the correctness properties of the algorithms. We also show how to construct failure-examples semi-systematically and discuss the geometry of the floating point implementation of the orientation predicate. We hope that the paper will be useful for teaching computational geometry. The paper comes with a companion web-page (`http://www.mpi-sb.mpg.de/~kettner/proj/NonRobust/`).

1 Introduction

The algorithms of computational geometry are designed for a machine model with exact real arithmetic. Substituting floating point arithmetic for the assumed real arithmetic may cause implementations to fail. Although this is well known, it is not common knowledge. There is no paper which systematically discusses what can go wrong and provides simple examples for the different ways in which floating point implementations can fail. Due to this lack of examples,

1. instructors of computational geometry have little material for demonstrating the inadequacy of floating point arithmetic for geometric computations,
2. students of computational geometry and implementers of geometric algorithms still underestimate the seriousness of the problem, and
3. researchers in our and neighboring disciplines, e.g., numerical analysis, still believe, that simple approaches are able to overcome the problem.

*Supported by ECG

†Max-Planck-Institut für Informatik, Saarbrücken, Germany, {ketter,mehlhorn}@mpi-sb.mpg.de

‡INRIA Sophia Antipolis, France, pion@inria.fr

§Otto-von-Guericke-Universität, Magdeburg, Germany, stschirr@isg.cs.uni-magdeburg.de

¶New York University, New York, USA, yap@cs.nyu.edu

In this paper, we study simple algorithms for two simple geometric problems, namely computing convex hulls and triangulations of point sets, and show how they can fail and explain why they fail when executed with floating point arithmetic. The convex hull $CH(S)$ of a set S of points in the plane is the smallest convex polygon containing S. A point $p \in S$ is called *extreme* in S if $CH(S) \neq CH(S \setminus p)$. The extreme points of S form the vertices of the convex hull polygon. Convex hulls can be constructed incrementally. One starts with three non-collinear points in S and then considers the remaining points in arbitrary order. When a point is considered and lies inside the current hull, the point is simply discarded. When the point lies outside, the tangents to the current hull are constructed and the hull is updated appropriately. We give a more detailed description of the algorithm in Section 4.1 and the complete `C++` program in the Appendix.

Figure 5 shows three points sets (we give the numerical coordinates of the points in Section 4.1) and the respective convex hulls computed by the floating point implementation of our algorithm. In each case the input points are indicated by small circles, the computed convex hull polygon is shown in green, and the extreme points are shown as red circles. The examples show that the implementation make make gross mistakes. It may leave out points which are clearly extreme, it may compute polygons which are clearly non-convex, and it may even run forever (the example is not shown here).

The first contribution of this paper is to provide a set of instances which make the floating point implementations fail in a disastrous way. The computed outputs do not resemble the correct outputs in any reasonable sense.

Our second contribution is to explain why these disasters happen. The correctness of geometric algorithms depends on geometric properties, e.g., a point lies outside a convex polygon if and only if it can see one of the edges. We give examples, for which a floating point implementation violates these properties: a point outside a convex polygon which sees no edge (in a floating point implementation of "sees") and a point not outside which sees some edges (in a floating point implementation of "sees"). We make it a point to give examples for all possible violations of the correctness properties of our algorithms.

Our third contribution is to show how difficult examples can be constructed systematically or at least semi-systematically. This should allow others to do similar studies for other algorithms.

We believe that the paper and its companion web page[1] will be useful in teaching computational geometry, and that even experts will find it surprising and instructing in how many ways and how badly even simple algorithms can be made to fail.

This paper is structured as follows. In Section 2 we discuss the ground rules for our experiments, in Section 3 we study the effect of floating point arithmetic on one of the most basic predicates of planar geometry, the orientation predicate, in Section 4 we discuss the incremental and the gift wrapping algorithm for planar convex hulls and in Section **??** we discuss an incremental algorithm for 3d Delaunay triangulations. Finally, Section 5 offers a short conclusion. The full paper of 22 pages is available at `www.mpi-sb.mpg.de/~mehlhorn/ftp/ClassRoomExamples.ps`. In this abstract the sections on gift wrapping and 3d Delaunay are omitted completely and the other sections are shortened.

Related Work: The literature contains a small number of documented examples of failures due to numerical imprecision, e.g., Forrest's seminal paper on implementing the point-in-polygon test [For79], Shewchuk's example for divide-and-conquer Delaunay triangulation [She97], Ramshaw's braided lines [MN99, Section 9.6.2], Schirra's example for convex hulls [MN99, Section 9.6.1], and the sweep

[1] The companion web page (`http://www.mpi-sb.mpg.de/~kettner/proj/NonRobust/`) contains the source code of all programs, the input files for all examples, and installation procedures. It allows the reader to redo our experiments and to perform further experiments.

line algorithm for line segment intersection and boolean operations on polygons [MN99, Sections 10.7.4 and 10.8.3].

2 Ground Rules for our Experiments

Our codes are written in `C++` and the results are reproducible on any platform compliant with IEEE floating-point standard 784 for double precision (see [Gol90]), and also with other programming languages. All programs and input data can be found on the companion web page. Numerical computations are based on IEEE arithmetic. In particular, numbers are machine floating point numbers (called *doubles* for short). Such numbers have the form $\pm m2^e$ where $m = 1.m_1m_2\dots m_{52}$ ($m_i = \{0,1\}$) is the mantissa in binary and e is the exponent satisfying $-1024 < e < 1024$. The results of arithmetic operations are rounded to the nearest double (with ties broken using some fixed rule).

Our numerical example data will be written in decimals (for human consumption). Such decimal values, when read into the machine, are internally represented by the nearest double. This conversion is automatic in modern compilers. We have made sure and have checked that our data can be represented exactly by doubles and their conversion to binary and back to decimal is the identity operation.

3 The Orientation Predicate

Three points p, q, and r in the plane either lie on a common line or form a left or right turn. The triple (p,q,r) forms a left (right) turn, if r lies left (right) of the line through p and q and oriented in the direction from p to q. Analytically, the orientation of the triple (p,q,r) is tantamount to the sign of a determinant:

$$orientation(p,q,r) = sign(\det \begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}), \tag{1}$$

where $p = (p_x, p_y)$, etc. We have $orientation(p,q,r) = +1$ (resp., -1, 0) iff the polyline (p,q,r) represents a left turn (resp., right turn, collinearity). Interchanging two points in the triple changes the sign of the orientation. We denote the x-coordinate of a point p also by $x(p)$.

We implement the orientation predicate in the straightforward way:

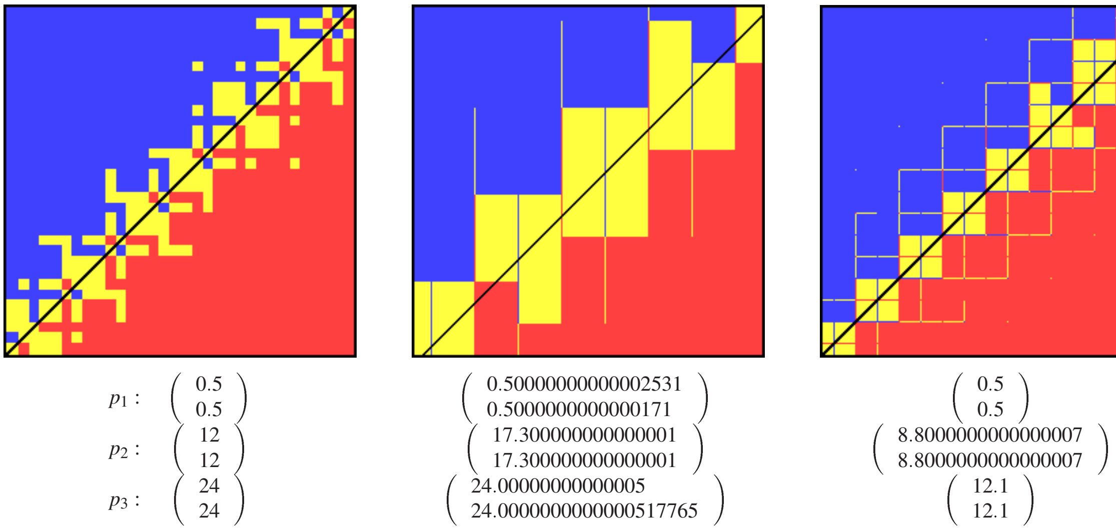
$$orientation(p,q,r) = sign((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)). \tag{2}$$

When the orientation predicate is implemented in this obvious way and evaluated with floating point arithmetic, we call it $float\_orient(p,q,r)$ to distinguish it from the ideal predicate. Since floating point arithmetic incurs round-off error, there are potentially three ways in which the result of *float_orient* could differ from the correct orientation:

- *rounding to zero*: we mis-classify a $+$ or $-$ as a 0;
- *perturbed zero*: 0 is mis-classified as $+$ or $-$;
- *sign inversion*: we mis-classify a $+$ as $-$ or vice-versa.

3.1 The Geometry of Float-Orientation

What is the geometry of *float_orient*, i.e., which triples of points are classified as left-turns, right-turns, or straight? The following type of experiment answers the question: We choose three points p_1, p_2,



(a) (b) (c)

Figure 1: The geometry of the float-orientation predicate. The figure shows the results *float_orient*$((x(p_1)+x\cdot u, y(p_1)+y\cdot u)+(x,y), p_2, p_3)$ for $0 \leq x, y \leq 255$, where u is the increment between adjacent floating point number in the considered range. The result is color coded: Yellow (red, blue, resp.) pixels represent collinear (negative, positive, resp.) orientation. The line through p_2 and p_3 is also shown.

and p_3 and then compute

$$\mathit{float\_orient}((x(p_1)+x\cdot u, y(p_1)+y\cdot u)+(x,y), p_2, p_3)$$

for $0 \leq x, y \leq 255$, where u is the increment between adjacent floating point numbers in the considered range; for example, $u = 2^{-53}$ if $x(p_1) = 1/2$ and $u = 4\cdot 2^{-53}$ if $x(p_1) = 2 = 4\cdot 1/2$. We visualize the resulting 256×256 array of signs as a 256×256 grid of colored pixels: A yellow (red, blue) pixel represents collinear (negative, positive, respectively) orientation. In the figures in this section we also indicate an approximation of the line through p_2 and p_3 by drawing all pixels black which have distance at most 2 from the line.

Figure 1(a) shows the result of our first experiment: We use the line defined by the points $p_2 = (12,12)$ and $p_3 = (24,24)$ and query it near $p_1 = (0.5, 0.5)$. We urge the reader to pause for a moment and to sketch what he/she expects to see. The authors expected to see a yellow band around the diagonal with nearly straight boundaries. Even for points with such simple coordinates the geometry of *float_orient* is quite weird: the set of yellow points (= the points classified as on the line) does not resemble a straight line and the sets of red or blue points do not resemble half-spaces. We even have points that change the side of the line, i.e., are lying left (right) of the line and being classified as right (left) of the line.

In Figures 1(b) and (c) we have given our base points p_i more complex coordinates by adding some digits behind the binary point. This enhances the cancellation effects in the evaluation of *float_orient* and leads to even more striking pictures. In (b), the red region looks like a step function at first sight. Note however, it is not monotone, has yellow rays extending into it, and red lines extruding from it. The yellow region (= on-region) forms blocks along the line. Strangely enough, these blocks are separated by blue and red lines. Finally, many points change sides. In Figure (c), we have yellow blocks of varying sizes along the diagonal, thin yellow and partly red lines extending into the blue region (similarly for the red region), red points (the left upper corners of the yellow structures extending into the blue region) deep inside the blue region, and isolated yellow points almost 100 units away from the diagonal.

Some Analysis: All three diagrams in Figure 1 exhibit block structure. We now explain why. Assume we keep y fixed and vary only x. We evaluate $\mathit{float\_orient}((x(p_1)+x\cdot u, y(p_1)+y\cdot u)+(x,y), p_2, p_3)$ for $0 \le x,y \le 255$, where u is the increment between adjacent floating point number in the considered range. Also $\mathit{orientation}(p,q,r) = \mathit{sign}((q_x-p_x)(r_y-p_y)-(q_y-p_y)(r_x-p_x))$. We incur round-off error in the additions/subtractions and also in the multiplications. Consider first one of the differences, say $q_x - p_x$. In (a), we have $q \approx 12$ and $p_x \approx 0.5$. Since $12 = 1\cdot 8+1\cdot 4+0\cdot 2+0\cdot 1$ has four digits before the binary point, we loose the last four bits of x in the subtraction, in other words, the result of the subtraction $q_x - p_x$ is constant for 2^4 consecutive values of x. Because of rounding to nearest, the intervals of constant value are $[8,23]$, $[24,39]$, $[40,55]$ .... Similarly, the floating point result of $r_x - p_x$ is constant for 2^5 consecutive values of x. Because of rounding to nearest, the intervals of constant value are $[16,47]$, $[48,69]$, ... Overlaying the two progressions gives intervals $[16,23]$, $[24,39]$, $[40,47]$, $[48,55]$, ... and this explains the structure we see in the rows of (a). We see short blocks of length 8, 16, 24, ... In (b) and (c), the situation is somewhat more complicated. It is again true that we have intervals for x, where there results of the subtractions are constant. However, since q and r have more complex coordinates, the relative shifts of these intervals are different and hence we see small and broad features.

Some Theory: We show that if all point coordinates are chosen from the range $[1/2,2]$, more generally, differ by a factor of at most two, then the only sign error is rounding to zero. According to Sterbenz's theorem [Ste74], floating point subtraction of two floating point numbers a and b is exact, if $1/2 \le a/b \le 2$ and so there will be no cancellation in the subtraction of point coordinates. Cancellation can only occur in the evaluation of the final expression of the form $cd - ef$. If $cd = ef$ then the floating point sign evaluation will return zero, since the double nearest to cd and ef is the same. If $cd \ge ef$, the result of computing cd in floating point arithmetic is at least as large as the result of computing ef in floating point arithmetic. Thus, floating point evaluation of $cd - ef$ results in non-negative number. We conclude that the only sign error is rounding to zero. Because of this analysis, we choose our point coordinates from a larger range in our examples.

4 The Convex Hull Problem

We discuss two simple convex hull algorithms, an incremental algorithm and the gift-wrapping method. In both cases, we describe the algorithm, state the underlying geometric assumptions, give instances which violate the assumptions when used with floating point arithmetic, and finally show which disastrous effects these violations may have on the result of the computation.

4.1 The Incremental Algorithm

The incremental algorithms maintains the *current convex hull CH* of the points seen so far. Initially, *CH* is formed by choosing three non-collinear points in *S*. It then considers the remaining points one by one. When considering a point r, it first determines whether r is outside the current convex hull polygon. If not, no action is required and r is discarded. Otherwise, the hull is updated by forming the tangents from r to *CH* and updating *CH* appropriately. The incremental paradigm is used in Andrew's variant [And79] of Graham's scan [Gra72] and also in the randomized incremental algorithm [CS89].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \le i \le k-1$ (indices are modulo k)

are the *edges* of the current hull. If $orientation(v_i, v_{i+1}, r) < 0$, we say r *sees* the edge (v_i, v_{i+1}), and say the edge (v_i, v_{i+1}) is *visible* from r. If $orientation(v_i, v_{i+1}, r) \leq 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, we have $k \geq 3$. The following two properties are key to the operation of the algorithm.

Lemma 1 (Property A) *A point r is outside CH iff there is an i such that the edge (v_i, v_{i+1}) is visible from r.*

Lemma 2 (Property B) *If r is outside CH, then the set of edges that are weakly visible from r forms a non-empty consecutive subchain; so does the set of edges that are not weakly visible from r.*

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense. E.g., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

Incremental Convex Hull Algorithm (Sketch):

Initialize L to the counter-clockwise triangle (a, b, c). Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $(v_i, \ldots, v_j)$ of edges that are weakly visible from r;
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r;
  end if
end for

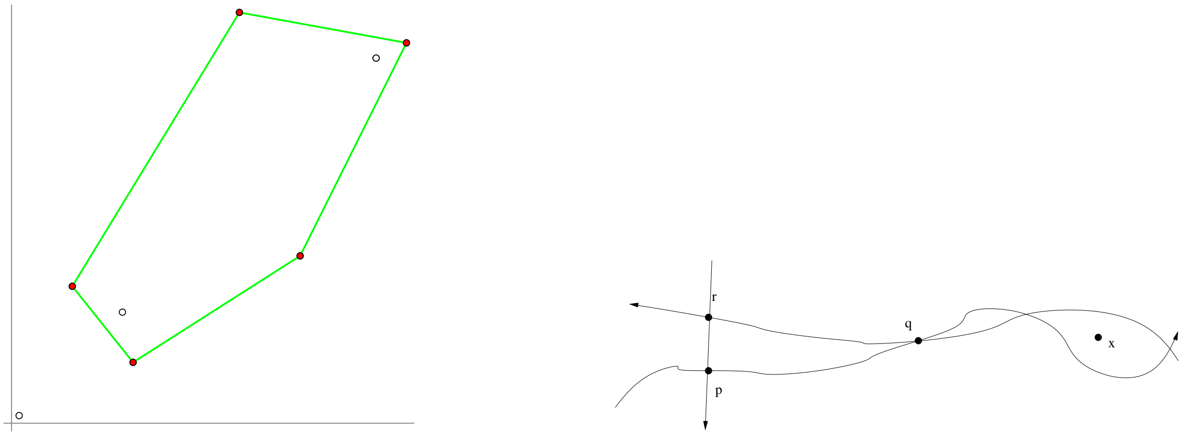
To turn the sketch into an algorithm, we provide additional information about the substeps.

1. How does one determine whether there is an edge e visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next item.
2. How does one identify the subsequence $(v_i, \ldots, v_j)$? Starting from the visible e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy").

We give a detailed implementation in the Appendix; it was used for all experiments. There are four logical ways to negate Properties A and B:

- **Failure (A_1):** A point outside the current hull sees no edges of the current hull.
- **Failure (A_2):** A point inside the current hull sees an edge of the current hull.
- **Failure (B_1):** A point outside the current hull sees all edges of the convex hull.
- **Failure (B_2):** A point outside the current hull sees a non-contiguous set of edges.

Failures (A_1) and (A_2) are equivalent to the negation of Property A. Similarly, Failures (B_1) and (B_2) are complete for Property B if we take (A_1) into account. Are all these failures realizable? We now affirm this.



(a) (b)

Figure 2: (a) The convex hull illustrating Failure (A_1). Observe the point in the lower left corner which is left out of the hull. (b) schematically indicates the ridiculous situation of a point outside the current hull and seeing no edge of the hull.

4.2 Single Step Failures

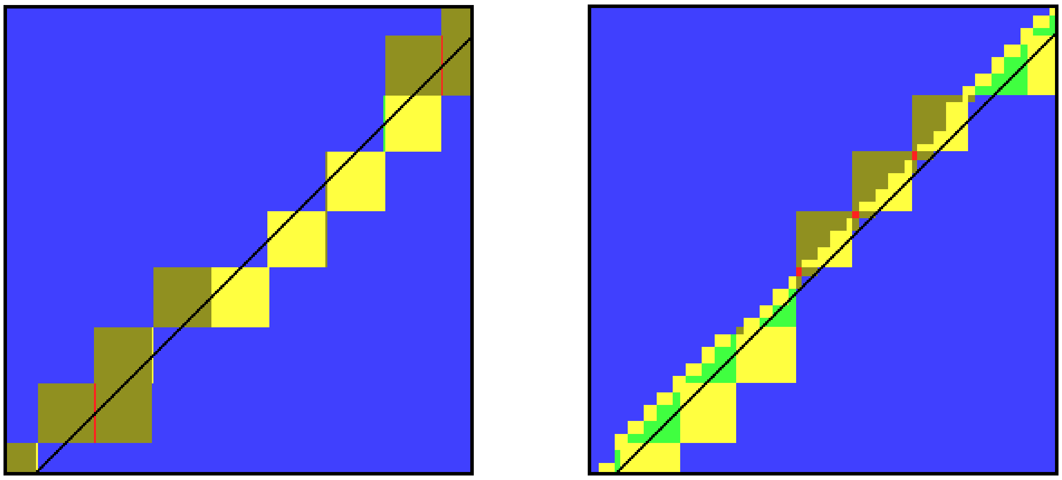
We give instances violating the correctness properties of the algorithm. More precisely, we give sequences p_1, p_2, p_3, ... of points such that the first three points form a counter-clockwise triangle (and *float_orient* correctly discovers this) and such that the insertion of some later point leads to a violation of a correctness property (in the computations with doubles). We also discuss how we arrived at the examples. All our examples involve nearly or truly collinear points; instances without nearly collinear or truly collinear points cause no problems in view of the error analysis of the preceding section (omitted in this extended abstract). Does this make our examples unrealistic? We believe not. Many point sets contain nearly collinear points or truly collinear points which become nearly collinear by conversion to floating point representation.

(A_1) A point outside the current hull sees no edge of the current hull: Consider the set of points below. Figure 2(a) shows the computed convex hull, where a point which is clearly extreme was left out of the hull.

$$\begin{aligned}
p_1 &= (7.3000000000000194, 7.3000000000000167) \\
p_2 &= (24.000000000000068, 24.000000000000071) \\
p_3 &= (24.00000000000005, 24.000000000000053) \\
p_4 &= (0.50000000000001621, 0.50000000000001243) \\
p_5 &= (8,4) \quad p_6 = (4,9) \quad p_7 = (15,27) \\
p_8 &= (26,25) \quad p_9 = (19,11)
\end{aligned}
\qquad
\begin{aligned}
&\mathit{float_orient}(p_1,p_2,p_3) > 0, \\
&\mathit{float_orient}(p_1,p_2,p_4) > 0, \\
&\mathit{float_orient}(p_2,p_3,p_4) > 0, \\
&\mathit{float_orient}(p_3,p_1,p_4) > 0 \ \ (!!).
\end{aligned}$$

What went wrong? The culprits are the first four points. They lie almost on the line $y = x$, and *float_orient* gives the results shown above. Only the last evaluation is wrong, indicated by "(!!)". Geometrically, these four evaluations say that p_4 sees no edge of the triangle (p_1, p_2, p_3). Figure 5(b) gives a schematic view of this ridiculous situation. The points $p_5, \ldots p_9$ are then correctly identified as extreme points and are added to the output sequence. However, the algorithm never recovers from the error made when considering p_4 and the result of the computation differs drastically from the correct hull.

We next explain how we arrived at the instance above. Intuition told us that an example (if it exists at all) would be a triangle with two almost parallel sides and with a query point near the wedge



p_1 :	$(17.300000000000001, 17.300000000000001)$	$(7.3000000000000194, 7.3000000000000167)$
p_2 :	$(24.000000000000068, 24.000000000000071)$	$(24.000000000000068, 24.000000000000071)$
p_3 :	$(24.00000000000005, 24.000000000000053)$	$(24.00000000000005, 24.000000000000053)$
p_4 :	$(0.5, 0.5)$	$(0.5, 0.5)$

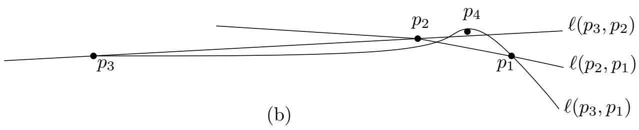
(a) (b)

Figure 3: The points (p_1, p_2, p_3) form a counter-clockwise triangle and we are interested in the classification of points $(x(p_4)+xu, y(p_4)+yu)$ in with respect to the edges (p_1, p_2) and (p_3, p_1) incident to p_1. The extensions of these edges are indistinguishable in the pictures and are drawn as a single black line. The red points do no "float-see" either one of the edges (Failure A_1). Points collinear with one of the edges are ocher, those collinear with both edges are yellow, those classified as seeing one but not the other edge are blue, and those seeing both edges are green. (a) Example starting from points in Figure 1. (b) Example that achieves "robustness" with respect to the first three points.

between the two nearly parallel edges. In view of Figure 1 such a point might be mis-classified with respect to one of the edges and hence would be unable to see any edge of the triangle. So we started with the points used in Figure 1(b), i.e., $p_1 \approx (17, 17)$, $p_2 \approx (24, 24) \approx p_3$, where we moved p_2 slightly to the right so as to guarantee that we obtain a counter-clockwise triangle. We then probed the edges incident to p_2 with points near p_1. Figure 3(a) visualizes the outcomes of the two relevant orientation tests. Each red pixel is a candidate for Failure (A_1). The example obtained in this way is not completely satisfactory, since some orientation tests on the initial triangle (p_1, p_2, p_3) were evaluating to zero.

We perturbed the example further, aided by visualizing the critical test *float_orient*(p_1, p_2, p_3) at the point p_1, until we found the examples shown in (b); by our error analysis, this test incurs the largest error and is hence most likely to return the incorrect result. The final example has the nice property that all possible *float_orient* tests on the first three points are correct. So this example is pretty much independent from any conceivable initialization an algorithm could use to create the first valid triangle. Figure 3(b) shows the two outcome of the two orientations tests for our final example.

(A_2) A point inside the current hull sees an edge of the current hull: Such examples are plenty. We take any counter-clockwise triangle and chose a fourth point inside the triangle but close to one of the edges. By Figure 1 there is the chance of sign reversal. A concrete example follows:



(b)

Figure 4: Schematics: The point p_4 sees all edges of the triangle (p_1, p_2, p_3).

$$\begin{aligned} p_1 &= (27.643564356435643, -21.881188118811881) \\ p_2 &= (83.366336633663366, 15.544554455445542) \\ p_3 &= (4, 4) \\ p_4 &= (73.415841584158414, 8.8613861386138595) \end{aligned}$$

$$\begin{aligned} &\mathit{float\_orient}(p_1, p_2, p_3) > 0 \\ &\mathit{float\_orient}(p_1, p_2, p_4) < 0 \text{ (!!)} \\ &\mathit{float\_orient}(p_2, p_3, p_4) > 0 \\ &\mathit{float\_orient}(p_3, p_1, p_4) > 0 \end{aligned}$$

The convex hull is correctly initialized to (p_1, p_2, p_3). The point p_4 is inside the current convex hull, but the algorithm incorrectly believes that p_4 can see the edge (p_1, p_2) and hence changes the hull to (p_1, p_4, p_2, p_3). This is a slightly non-convex polygon.

(B_1) A point outside the current hull sees all edges of the convex hull Intuition told us that an example (if it exists) would consist of a triangle with one angle close to π and hence three almost parallel sides. Where should one place the query point. We first placed it in the extension of the three parallel sides and quite a distance away from the triangle. This did not work. The choice which worked is to place the point near one of the sides so that it could see two of the sides and "float-see" the third. Figure 4 illustrates this choice. A concrete example follows:

$$\begin{aligned} p_1 &= (200, 49.200000000000003) \\ p_2 &= (100, 49.600000000000001) \\ p_3 &= (-233.33333333333334, 50.93333333333333) \\ p_4 &= (166.66666666666669, 49.333333333333336) \end{aligned}$$

$$\begin{aligned} &\mathit{float\_orient}(p_1, p_2, p_3) > 0 \\ &\mathit{float\_orient}(p_1, p_2, p_4) < 0 \\ &\mathit{float\_orient}(p_2, p_3, p_4) < 0 \\ &\mathit{float\_orient}(p_3, p_1, p_4) < 0 \text{ (!!)} \end{aligned}$$

The first three points form a counter-clockwise oriented triangle and according to *float_orient*, the algorithm believe that p_4 can see all edges of the triangle. What will our algorithm do? It depends on the implementation details. If the algorithm first searches for an invisible edge, it will search forever and never terminate. If it deletes points on-line from L it will crash or compute nonsense depending on the details of the implementation.

(B_2) A point outside the current hull sees a non-contiguous set of edges Consider the following points:

$$\begin{aligned} p_1 &= (0.50000000000001243, 0.50000000000000189) \\ p_2 &= (0.50000000000001243, 0.50000000000000333) \\ p_3 &= (24.00000000000005, 24.000000000000053) \\ p_4 &= (24.000000000000068, 24.000000000000071) \\ p_5 &= (17.300000000000001, 17.300000000000001) \end{aligned}$$

$$\begin{aligned} &\mathit{float\_orient}(p_1, p_4, p_5) < 0 \text{ (!!)} \\ &\mathit{float\_orient}(p_4, p_3, p_5) > 0 \\ &\mathit{float\_orient}(p_3, p_2, p_5) < 0 \\ &\mathit{float\_orient}(p_2, p_1, p_5) > 0 \end{aligned}$$

Inserting the first four points results in the correct convex quadrilateral (p_1, p_4, p_3, p_2) in counter-clockwise order. The last point p_5 sees only the edge (p_3, p_2) and none of the other three. However, *float_orient* makes p_5 see also the edge (p_1, p_4). The subsequences of visible and invisible edges are not contiguous. Since the falsely classified edge (p_1, p_4) comes first, our algorithm inserts p_5 at this edge, removes no other vertex, and returns a polygon that has self-intersections and is not simple.

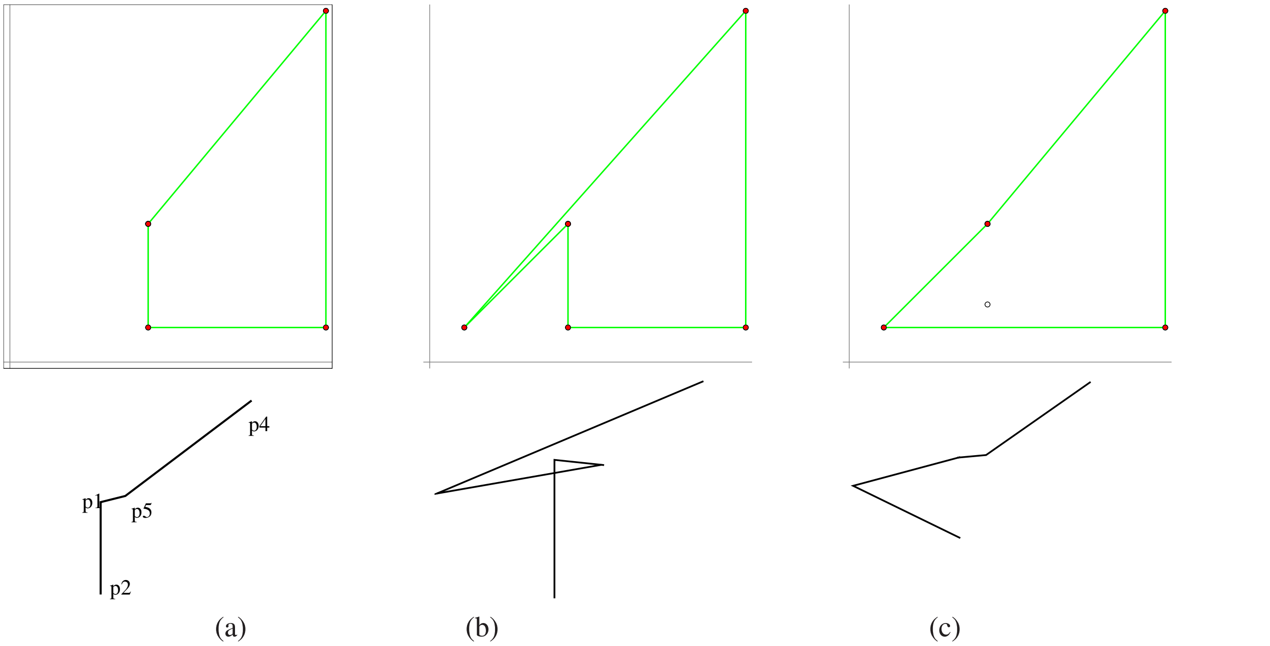


Figure 5: (a) The point in the left upper corner is really two points and we have a concave corner at one of them. This is illustrated in the schematic sketch. The point p_6 sees two edges, which are however not consecutive. Depending on which one is actually chosen by the algorithm, we obtain the hulls shown in (b) or (c). We refer the reader to the text for further explanations. The figures show the coordinate axes for orientation.

4.3 Global Effects

By now, we have seen examples which cover the negation space of the correctness properties on the incremental algorithm and we have seen the effect of an incorrect orientation test for a single update step. We next study global effect. The goal is to refute the myth that the algorithm will always compute an approximation of the true convex hull.

The algorithm computes a convex polygon, but misses some of the extreme points: We have already seen such an example in Failure (A_1). We can modify this example so that the ratio of the areas of the true hull and the computed hull becomes arbitrarily large. We do as in Failure (A_1), but move the fourth point to infinity. The true convex hull has four extreme points. The algorithm misses q_4.

$$
\begin{array}{lcl}
q_1 & = & (0.10000000000000001, 0.10000000000000001) \\
q_2 & = & (0.20000000000000001, 0.20000000000000004) \\
q_3 & = & (0.79999999999999993, 0.80000000000000004) \\
q_4 & = & (1.267650600228229 \cdot 10^{30}, 1.2676506002282291 \cdot 10^{30})
\end{array}
\qquad
\begin{array}{l}
\mathit{float\_orient}(q_1,q_2,q_3) < 0 \\
\mathit{float\_orient}(q_1,q_2,q_4) = 0 \ (!!) \\
\mathit{float\_orient}(q_2,q_3,q_4) = 0 \ (!!) \\
\mathit{float\_orient}(q_3,q_1,q_4) > 0
\end{array}
$$

The algorithm does not terminate or crashes: We have seen this behavior in Failure (B_1).

The algorithm computes a non-convex polygon: We have already given such an example in Failure (A_2). However, this failure is not visible to the naked eye. We next give examples where non-convexity is visible to the naked eye. We consider the points:

$$
\begin{array}{lcllcl}
p_1 & = & (24.00000000000005, 24.0000000000000053) & p_2 & = & (24.0, 10.0) \\
p_3 & = & (54.85, 6.0) & p_4 & = & (54.850000000000357, 61.000000000000121) \\
p_5 & = & (24.000000000000068, 24.000000000000071) & p_6 & = & (6, 6)\,.
\end{array}
$$

After the insertion of p_1 to p_4, we have the convex hull (p_1, p_2, p_3, p_4). This is correct. The point p_5 lies inside the convex hull of the first four points; however, *float_orient*$(p_4, p_1, p_5) < 0$. Thus p_5 is inserted between p_4 and p_1. and we obtain $(p_1, p_2, p_3, p_4, p_5)$. However, this error is not visible yet to the eye, see Figure 5(a).

The point p_6 sees the edges (p_4, p_5) and (p_1, p_2), but does not see the edge (p_5, p_1). All of this is correctly determined by *float_orient*. Consider now the insertion process for point p_6. Depending on where we start the search for a visible edge, we will either find the edge (p_4, p_5) or the edge (p_1, p_2). In the former case, we insert p_6 between p_4 and p_5 and obtain the polygon shown in (b). It is visibly non-convex and has a self-intersection.In the latter case, we insert p_6 between p_1 and p_2 and obtain the polygon shown in (c). It is visibly non-convex.

Of course, in a deterministic implementation, we will see only one of the errors. In order to produce both errors with the implementation we insert an auxiliary point before p_6 in (c). This point sees only the edge (p_1, p_2) and makes sure that we start the search for a visible edge at the right place when p_6 in inserted.

5 Conclusion

We provided instances which cause floating point implementations of three basic geometric algorithms to fail. Our instances make the algorithms fail in many different ways. We showed how to construct such instances semi-systematically. We hope that our paper and its companion web page will be useful for classroom use and that it will alert students and researchers to the intricacies of implementing geometric algorithms.

What can be done to overcome the robust problem of floating point arithmetic? There are essentially three approaches: (1) make sure that the implementation of the orientation predicate always returns the correct result or (2) change the algorithm so that it can cope with the floating point implementation of the orientation predicate and still computes something meaningful or (3) perturb the input so that the floating point implementation is guaranteed to produce the correct result on the perturbed input. The first approach is the most general and is known under the the exact geometric computation (EGC) paradigm and was first used by Jünger, Reinelt and Zepf [JRZ91] and Karasick, Lieber and Nackmann [KLN91]. For detailed discussions, we refer the reader to [Yap03] and Chapter 9 of [MN99]. The EGC approach has been adopted for the software libraries LEDA and CGAL and other successful implementations. The second and third approaches have been successfully applied to a number of geometric problems, see for example [Mil89, FM91, LM90, DSB92, SIII00, HS98]. But in the second approach the interpretation of "meaningful" is a crucial and difficult problem. The third approach should in principle be applicable to all problems with purely numerical input. There are also suggested approaches, e.g., epsilon-tweaking, which do not work. Epsilon-tweaking simply activates rounding to zero, both for correct and mis-classified orientations. E.g., it is now more likely for points outside the current hull not to see any edges because of enforced collinearity.

References

[And79] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979.

[CS89] K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, II. *Journal of Discrete and Computational Geometry*, 4:387–421, 1989.

[DSB92] T. K. Dey, K. Sugihara, and C. L. Bajaj. Delaunay triangulations in three dimensions with finite precision arithmetic. *Comput. Aided Geom. Design*, 9:457–470, 1992.

[FGK+00] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.

[FM91] S. Fortune and V.J. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry (SCG'91)*, pages 334–341. ACM Press, 1991.

[For79] A. R. Forrest. On the rendering of surfaces. volume 13, pages 253–259, August 1979.

[Gol90] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1990.

[Gra72] R.L. Graham. An efficient algorithm for determining the convex hulls of a finite point set. *Information Processing Letters*, 1:132–133, 1972.

[HS98] Halperin and Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *CGTA: Computational Geometry: Theory and Applications*, 10, 1998.

[JRZ91] M. Jünger, G. Reinelt, and D. Zepf. Computing correct delaunay triangulations. *Computing*, 47:43–49, 1991.

[KLN91] M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[LM90] Z. Li and V.J. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In *Proceedings of the 6th Annual ACM Symposium on Computational Geometry (SCG'90)*, pages 235–243. ACM Press, 1990.

[Mil89] V.J. Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In Kurt Mehlhorn, editor, *Proceedings of the 5th Annual ACM Symposium on Computational Geometry (SCG'89)*, pages 197–207. ACM Press, 1989.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.

[She97] J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[SIII00] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementation - an approach to robust geometric algorithms. *Algorithmica*, 27(1):5–20, 2000.

[Ste74] P.H. Sterbenz. *Floating Point Computation*. Prentice Hall, 1974.

[Yap03] C. K. Yap. Robust geometric computation. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41. CRC Press LLC, Boca Raton, FL, 2nd edition, 2003.

APPENDIX A: Implementation

We describe our C++ reference implementation of our simple incremental algorithm. We give the details necessary to reproduce our results, for example, the exact parameter order in the predicate calls, but we omit details of the startup phase when we search for the initial three non-collinear points and the circular list data structure. We offer the full working source code based on CGAL [FGK+00], all the point data sets, and the images from the analysis on our web-site `http://www.mpi-sb.mpg.de/~kettner/proj/NonRobust/` for reference.

We use our own plain conventional C++ point type. Worth mentioning are equality comparison and lexicographic order used to find extreme points among collinear points in the startup phase.

```
struct Point { double x, y; };
```

The orientation test returns $+1$ if the points p, q, and r make a leftturn, it returns zero if they are collinear, and it returns -1 is they form a rightturn. We implement the orientation test as explained above with p as pivot point. Not shown here, but we make sure that all intermediate results are represented as 64 bit doubles and not as 80 bit extended doubles as is might happen, e.g., on Intel platforms.

```
int orientation( Point p, Point q, Point r) {
    return sign((q.x-p.x) * (r.y-p.y) - (q.y-p.y) * (r.x-p.x));
}
```

For the initial three non-collinear points we scan the input sequence and maintain its convex hull of up to two extreme points until we run out of input points or we find a third extreme point for the convex hull. From there on we scan the remaining points in our main `convex_hull` function as shown below.

The circular list used in our implementation is self explaining in its use. We assume a Standard Template Library (STL) compliant interface and extend it with circulators, a concept similar to STL iterators that allow the circular traversal in the list without any past-the-end position using the increment and decrement operators. In addition, we assume a function that can remove a range in the list specified by two non-identical circulator positions.

Our main `convex_hull` function shown below has a conventional iterator-based interface like other STL algorithms. It computes the extreme points in counterclockwise order of the 2d convex hull of the points in the iterator range `[first,last)`. It uses internally the circular list `hull` to store the current extreme points and copies this list to the output iterator `result` at the end of the function. It also returns the modified `result` iterator.

```
template <typename ForwardIter, typename OutputIter>
OutputIter convex_hull( ForwardIter first, ForwardIter last, OutputIter result){
    typedef std::iterator_traits<ForwardIter>      Iterator_traits;
    typedef typename Iterator_traits::value_type  Point;
    typedef Circular_list<Point>                   Hull;
    typedef typename Hull::circulator              Circulator;

    Hull hull; // extreme points in counterclockwise (ccw) orientation
    // first the degenerate cases until we have a proper triangle
    first = find_first_triangle( first, last, hull);
    while ( first != last) {
        Point p = *first;
```

```
        // find visible edge in circular list of vertices of current hull
        Circulator c_source = hull.circulator_begin();
        Circulator c_dest = c_source;
        do {
            c_source = c_dest++;
            if ( orientation( *c_source, *c_dest, p) < 0) {
                // found visible edge, find ccw tangent
                Circulator c_succ = c_dest++;
                while ( orientation( *c_succ, *c_dest, p) <= 0)
                    c_succ = c_dest++;
                // find cw tangent
                Circulator c_pred = c_source--;
                while ( orientation( *c_source, *c_pred, p) <= 0)
                    c_pred = c_source--;
                // c˙source is the first point visible, c˙succ the last
                if ( ++c_pred != c_succ)
                    hull.circular_remove( c_pred, c_succ);
                hull.insert( c_succ, p);
                break; // we processed all visible edges
            }
        } while ( c_source != hull.circulator_begin());
        ++first;
    }
    return std::copy( hull.begin(), hull.end(), result);
}
```