Classroom Examples of Robustness Problems in Geometric Computations*

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

June 17, 2008

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 are no concrete examples with a comprehensive documentation of what can go wrong and why. In this paper, we provide a case study of what can go wrong and why. For our study, we have chosen two simple algorithms which are often taught, an algorithm for computing convex hulls in the plane and an algorithm for computing Delaunay triangulations in space. We give examples that make the algorithms fail in many different ways. We also show how to construct such examples systematically and discuss the geometry of the floating-point implementation of the orientation predicate. We hope that our work will be useful for teaching computational geometry.

1 Introduction

The algorithms of computational geometry are designed for a machine model with exact real arithmetic. It is well-known that substituting floating-point arithmetic for the assumed real arithmetic may cause implementations to fail. However, there are no concrete comprehensive examples. There is neither a paper nor a textbook that 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,

> instructors of computational geometry have little material for demonstrating the inadequacy of floating-point arithmetic for geometric computations,
>
> students of computational geometry and implementers of geometric algorithms still have to learn about the seriousness of robustness problems by experiencing the difficulties while programming.

*Partially supported by the IST Programme of the EU under Contract No IST-2000-26473, Effective Computational Geometry for Curves and Surfaces (ECG). A preliminary version of this paper appeared at ESA 2004, LNCS 3221, pages 702 – 713.

†MPI für Informatik, Saarbrücken, {kettner,mehlhorn}@mpi-inf.mpg.de

‡INRIA Sophia Antipolis, Sylvain.Pion@sophia.inria.fr

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

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

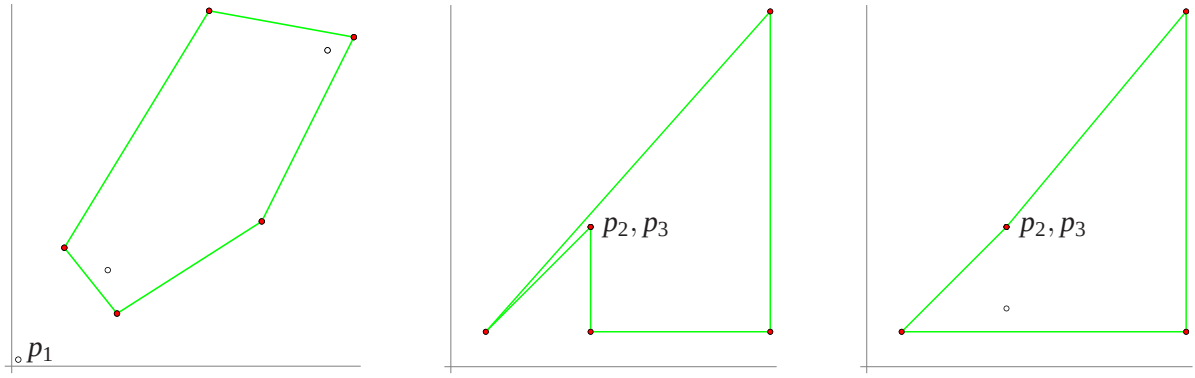


Figure 1: Results of a convex hull algorithm using double-precision floating-point arithmetic with the coordinate axes drawn to give the reader a frame of reference. The algorithm makes gross mistakes (from left to right): The clearly extreme point p_1 is left out. The convex hull has a large concave corner with a (non-visible) self intersection near p_2 and p_3, which are close together. The convex hull has a clearly visible concave chain (and no self-intersection). Details on these examples are explained in Section 4.

In this paper, we provide a case study of what can go wrong and why with geometric algorithms when executed with floating-point arithmetic naïvely. For our study, we have chosen two simple algorithms which are often taught, an algorithm for computing convex hulls in the plane and an algorithm to compute Delaunay triangulations in space.

The convex hull $CH(S)$ of a finite 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 1 shows point sets (we give the numerical coordinates of the points in Section 4) 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 alleged extreme points are shown as filled red circles. The examples show that the implementation may make gross mistakes. It may leave out points that are clearly extreme, it may compute polygons that are clearly non-convex, and it may even run forever.

The first contribution of this paper is to provide a set of instances that make the floating-point implementations fail, often in disastrous ways. The computed results do not resemble the correct results 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 from the outside. We give examples, for which a floating-point implementation violates these properties: a point outside a convex polygon that sees no edge and a point not outside that sees some edges (both in a floating-point implementation of "sees"). We give examples for all possible violations of the correctness properties of our convex hull algorithms.

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

We believe that the paper and its companion web page will be useful in teaching computational geometry, and that even experts will find it surprising and instructive in how many ways and how badly even simple algorithms can be made to fail. The companion web page[1] contains the source code of all programs, the input files for all examples, and installation procedures. It allows the reader to perform our and further experiments.

Numerical analysts are well aware of the pitfalls of floating point computation [For70]. Forsythe's paper and many numerical analysis textbooks, see for example [DH91, page 9], contain instructive examples of how popular algorithms, e.g., Gaussian elimination, can fail when used with floating point arithmetic. These examples have played a guiding role in the development of robust numerical methods. Our examples are in the same spirit, but concentrate on the geometric consequences of approximate arithmetic. While sophisticated machinery was developed for making numerical computations reliable over the past 50 years, a corresponding machinery for geometric computation does not yet exist to the same extent. However, significant progress was made over the past 15 years and we point the reader to approaches to reliable geometric computing in the conclusions: the exact computation paradigm, algorithms with reduced arithmetic demand, approximate algorithms with a correctness proof in floating-point arithmetic, and perturbation methods. In our recent courses on geometric computing, we have used the warning negative examples of this paper to raise student awareness for the problem and then discussed the approaches mentioned in the conclusions.

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 algorithm for planar convex hulls and in Section 5 we briefly discuss an incremental algorithm for 3d Delaunay triangulations. We provide a discussion of failures of the gift-wrapping in an accompanying report available on the companion web page of our paper. In Section 6 we discuss two frequently suggested simple approaches for making the planar convex hull algorithm more robust and argue that they fail. Finally, Section 7 offers a short conclusion and points to approaches to reliable geometric computation.

Related Work: The literature contains a small number of documented failures due to numerical imprecision, e.g., Forrest's seminal paper on implementing the point-in-polygon test [For85], Fortune's example for a variant of Graham's scan [For90][2], 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 line algorithm for line segment intersection and boolean operations on polygons [MN99, Sections 10.7.4 and 10.8.4].

2 Ground Rules for our Experiments

Our codes are written in C++ and the results are reproducible on any platform compliant with IEEE Std 754-1985 floating-point standard for double precision (see [Gol91, IEE87]), 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, we study machine floating-point numbers, called *doubles*, that are ubiquitous in scientific and geometric computing. Such numbers have the form $\pm m2^e$ where $m = 1.m_1m_2\ldots m_{52}$ ($m_i \in \{0,1\}$) is the mantissa in binary and e is the exponent satisfying $-1023 < e < 1024$.[3] The results of arithmetic operations are rounded to the nearest double

[1] `http://www.mpi-inf.mpg.de/departments/d1/ClassroomExamples/`

[2] The example is not contained in [For89]

[3] We ignore here so called *denormalized* numbers that play no role in our experiments and arguments.

(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. We have made sure that our data can be safely converted in this manner, i.e., conversion to binary and back to decimal is the identity operation. However, the C++ standard library does not provide sufficient guarantees and we offer additionally the binary data in little-endian format on the accompanying web page.

The programs were developed with the help of CGAL, the *Computational Geometry Algorithms Library*,[4] and LEDA, the *Library of Efficient Data Types and Algorithms*[5] [KN04, FGK+00, MN99]. To simplify the use in the classroom, the convex hull algorithms presented in this paper can be used independently of these (and other) libraries.

3 Planar Orientation Predicate

Three points $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$ 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 to the 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:

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

We have $\mathit{orientation}(p,q,r) = +1$ (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 implement the orientation predicate in the straightforward way:

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

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 errors, 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*: we mis-classify 0 as $+$ or $-$;
- *sign inversion*: we mis-classify a $+$ as $-$ or vice-versa.

3.1 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 collinear? The following type of experiment partially answers the question: We choose three points p, q, and r and then compute *float_orient* for points in the floating-point neighborhood of p and the remaining points q and r. More precisely, let u_x be the increment between adjacent floating-point numbers in the range right of p_x; for example, $u_x = 2^{-53}$ if $p_x = \frac{1}{2}$ and $u_x = 4 \cdot 2^{-53}$ if $p_x = 2 = 4 \cdot \frac{1}{2}$. Analogously, we define u_y. We consider

$$\mathit{float\_orient}((p_x + X u_x, p_y + Y u_y), q, r)$$

[4] http://www.cgal.org/

[5] http://www.algorithmic-solutions.com/enleda.htm

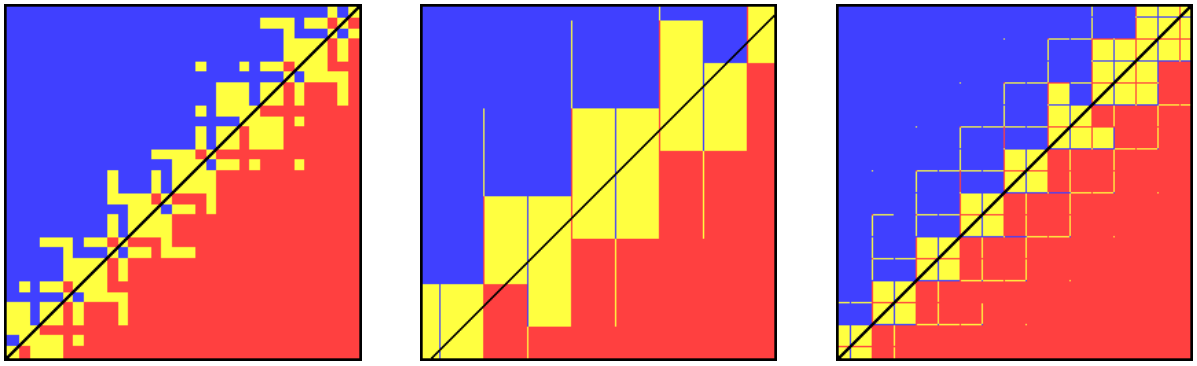


Figure 2: The weird geometry of the float-orientation predicate: The figure shows the results of *float_orient*$(p_x + Xu_x, p_y + Yu_y, q, r)$ for $0 \leq X, Y \leq 255$, where $u_x = u_y = 2^{-53}$ is the increment between adjacent floating-point numbers in the considered range. The result is color coded:[5] Yellow (red, blue, resp.) pixels represent collinear (negative, positive, resp.) orientation. The line through q and r is shown in black.

for $0 \leq X, Y \leq 255$. 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 exact line through q and r in black.

Figure 2(a) shows the result of our first experiment: We use the line defined by the points $q = (12, 12)$ and $r = (24, 24)$ and query it near $p = (0.5, 0.5)$. 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 of the line and being classified as right of the line and vice versa.

In Figures 2(b) and (c) we have given our base points coordinates with more bits of precision 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 (= collinear-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.

All diagrams in Figure 2 exhibit block structure. We now explain why: We focus on one dimension,

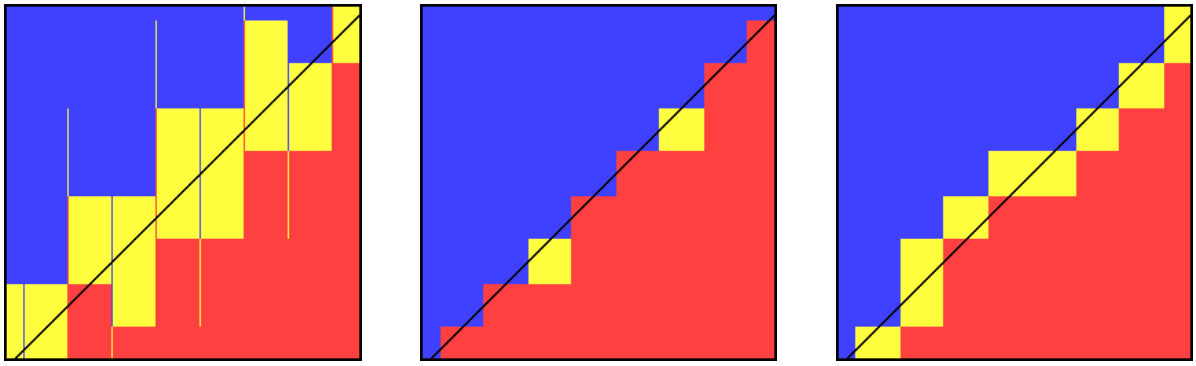
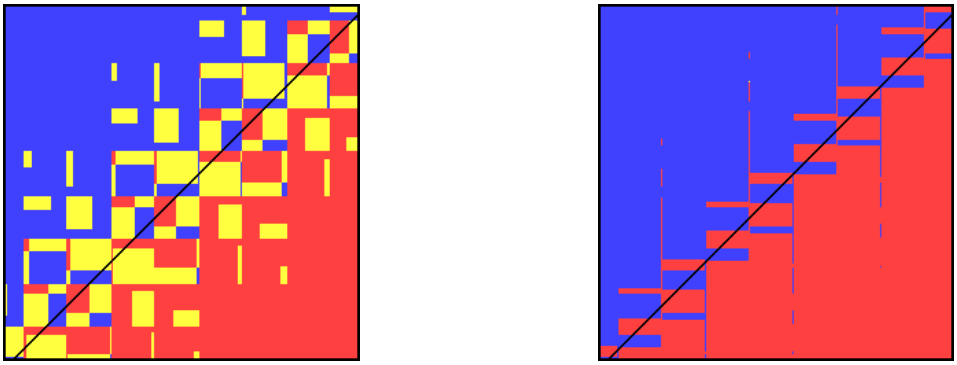


Figure 3: We repeat the example from Figure 2(b) and show the result for all three distinct choices for the pivot; namely p on the left, q in the middle, and r on the right. All figures exhibit sign reversal.

i.e., assume we keep Y fixed and vary only X. We evaluate $\mathit{float\_orient}((p_x + Xu_x, p_y + Yu_y), q, r)$ for $0 \leq X \leq 255$, where $u_x = u_y$ is the increment between adjacent floating-point numbers in the considered range. Recall that $\mathit{orientation}(p,q,r) = \mathit{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))$. We incur round-off errors in the additions/subtractions and also in the multiplications. Consider first one of the differences, say $q_x - p_x$. In (a), we have $q_x = 12$ and $p_x \approx 0.5$. Since 12 has four binary digits, we lose 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 (a). In (b) and (c), the situation is somewhat more complicated. It is again true that we have intervals for X, where the 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 narrow and broad features.

Next we show that if all point coordinates 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 $\frac{1}{2} \leq \frac{a}{b} \leq 2$, 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 $cd \geq ef$, 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, the floating-point evaluation of $cd - ef$ results in a 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.

Choice of a Pivot Point: The orientation predicate is the sign of a three-by-three determinant and this determinant may be evaluated in different ways. In *float_orient* as defined above we use the point p as the *pivot*, i.e., we subtract the row representing the point p from the other rows and reduce the problem to the evaluation of a two-by-two determinant. Similarly, we may choose one of the other points as the pivot. Figure 3 displays the effect of the different choices of the pivot point on the example of Figure 2(b). The choice of the pivot makes a difference, but nonetheless the geometry remains non-trivial and sign reversals happen for all three choices.



(a) **(b)**

Figure 4: Examples of the impact of extended double arithmetic. We repeat the example from Figure 2(b) with different implementations of the orientation test: **(a)** We evaluate $(q_x - p_x)(r_y - p_y)$ and $(q_y - p_y)(r_x - p_x)$ in extended double arithmetic, convert their values to double precision, and compare them. **(b)** We evaluate $sign((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))$ in extended double arithmetic. For both experiments, we used $u_x = u_y = 2^{-53}$, the same as for the regular double precision examples in Figure 2. Note that there are no collinearities (yellow points) reported in **(b)**.

Based on floating-point error-bound estimates one can conclude that the center point w.r.t. the x-coordinate (or equivalently the y-coordinate) is the best choice for the pivot. This is implemented in the orientation test used by Fortune [For89]. However, the necessary conditional branching could impair performance significantly. If one is willing to invest that time, one could also think of using an exact implementation scheme based on floating-point filter techniques, e.g. [FvW96, She97], see [ST06] for results of an experimental comparison. Further details are beyond the scope of this paper.

Extended Double Precision: Some architectures, for example, Intel Pentium processors, offer IEEE extended double precision with a 64 bit mantissa in an 80 bit representation. Does this additional precision help? Not really, as the examples in Figure 4 suggest. One might argue that the number of misclassified points decreases, but the geometry of *float_orient* remains fractured and exploitable for failures similar to those that we develop below for double precision arithmetic.

4 Planar Convex Hull Problem

We discuss a simple planar convex hull algorithm that computes the convex hull incrementally. We describe the algorithm, state the underlying geometric assumptions, give instances that 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 Incremental Convex Hull Algorithm

The incremental algorithm 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, 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 [And79] and other variants 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}), $0 \leq i \leq k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leq 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geq 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside *CH* iff r can see an edge of *CH*.

Property B. If r is outside *CH*, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If (v_i, v_{i+1}), ..., (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, i.e., 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 a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. 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 $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

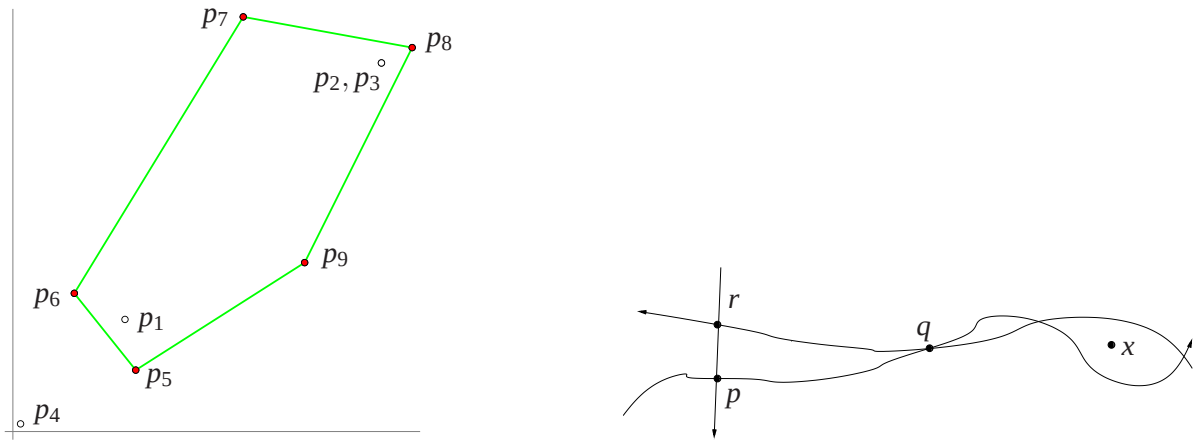
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.

2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we 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"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in the appendix; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:



(a) **(b)**

Figure 5: **(a)** The convex hull illustrating Failure A_1: The point p_4 in the lower left corner is left out of the hull. **(b)** Schematic view indicating the impossible situation of a point outside the current hull and seeing no edge of the hull: x lies to the left of all sides of the triangle (p,q,r).

- **Failure A_1:** A point outside the current hull sees no edge 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.

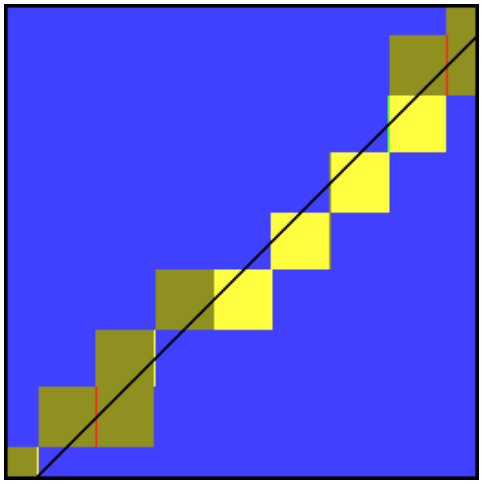
Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

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 *float_orient*). We also discuss how we arrived at the examples. All our examples involve nearly or truly collinear points; in the view of a standard rounding-error analysis sufficiently non-collinear points would not cause any problems. 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.

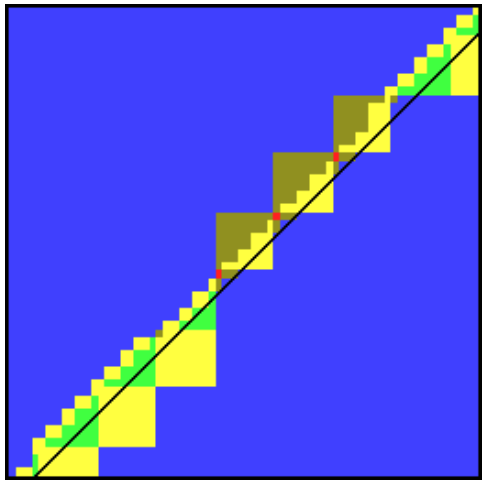
Failure A_1: A point outside the current hull sees no edge of the current hull: Consider the set of points below. Figure 5(a) shows the computed convex hull, where a point that is clearly extreme was left out of the hull.

$p_1 = (\ 7.3000000000000194,\ \ 7.3000000000000167\ )$ $\quad$ *float_orient*$(p_1,p_2,p_3) > 0$
$p_2 = (24.000000000000068,\ \ 24.000000000000071\ )$ $\quad$ *float_orient*$(p_1,p_2,p_4) > 0$
$p_3 = (24.00000000000005,\ \ 24.000000000000053\ )$ $\quad$ *float_orient*$(p_2,p_3,p_4) > 0$
$p_4 = (\ 0.50000000000001621,\ 0.50000000000001243)$ $\quad$ *float_orient*$(p_3,p_1,p_4) > 0$ (??)
$p_5 = (\ 8,\ 4)\quad p_6 = (\ 4,\ 9)\quad p_7 = (15,27)$
$p_8 = (26,25)\quad p_9 = (19,11)$



p_1 : $(17.300000000000001, 17.300000000000001)$
p_2 : $(24.000000000000068, 24.000000000000071)$
p_3 : $(24.00000000000005, 24.000000000000053)$
p_4 : $(0.50000000000000711, 0.5)$

(a)



$(7.3000000000000194, 7.3000000000000167)$
$(24.000000000000068, 24.000000000000071)$
$(24.00000000000005, 24.000000000000053)$
$(0.50000000000000355, 0.5)$

(b)

Figure 6: The points (p_1, p_2, p_3) form a counter-clockwise triangle and we are interested in the classification of points $(x(p_4) + Xu_x, y(p_4) + Yu_y)$ 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 not "float-see" either one of the edges (Failure A_1). These are the points we were looking for. The 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 2. **(b)** Example that achieves "invariance" with respect to permutation of the first three points.

What went wrong? Let us look at 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 impossible situation. The points $p_5, \ldots, p_9$ are then correctly identified as extreme points and are added to the hull. 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 defined by the two nearly parallel edges. In view of Figure 2 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 2(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_1 with points p_4 in and near the wedge formed by these edges. Figure 6(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 was 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 *float_orient*(p_1, p_2, p_3), until we found the example shown in (b). The final example has the nice property that all possible *float_orient* tests on

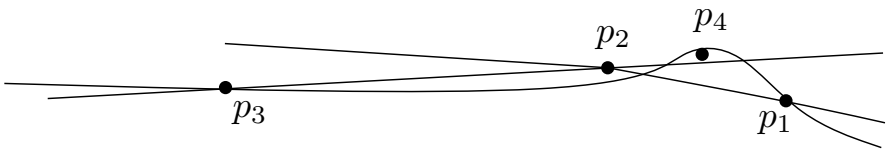


Figure 7: Schematic view of Failure B_1: The point p_4 sees all edges of the triangle (p_1, p_2, p_3).

the first three points are correct. So this example is independent from any conceivable initialization an algorithm could use to create the first valid triangle. Figure 6(b) shows the outcomes of the two orientations tests for our final example.

Failure A_2: A point inside the current hull sees an edge of the current hull: We take any counter-clockwise triangle and choose a fourth point inside the triangle but close to one of the edges. By Figure 2 there is the chance of sign reversal. A concrete example follows:

$$\begin{array}{lcll} p_1 & = & (27.643564356435643, & -21.881188118811881\ ) \\ p_2 & = & (83.366336633663366, & 15.544554455445542\ ) \\ p_3 & = & (\ 4.0, & 4.0\ ) \\ p_4 & = & (73.415841584158414, & 8.8613861386138595) \end{array} \qquad \begin{array}{l} \textit{float\_orient}(p_1, p_2, p_3) > 0 \\ \textit{float\_orient}(p_1, p_2, p_4) < 0\ (??) \\ \textit{float\_orient}(p_2, p_3, p_4) > 0 \\ \textit{float\_orient}(p_3, p_1, p_4) > 0 \end{array}$$

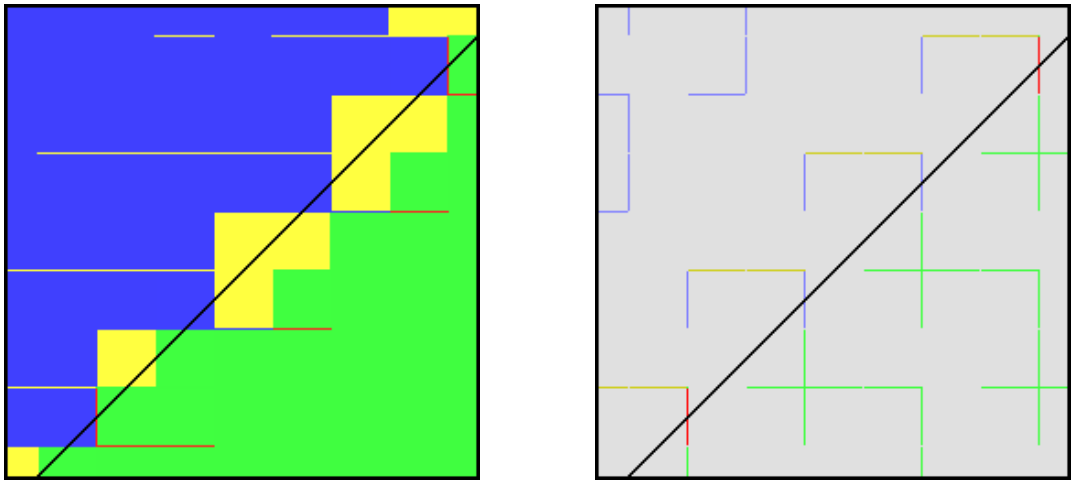
The convex hull is correctly initialized to (p_1, p_2, p_3). The point p_4 is inside the current convex hull, but the algorithm incorrectly believes that p_4 can see the edge (p_1, p_2) and hence changes the hull to (p_1, p_4, p_2, p_3), a slightly non-convex polygon.

Failure 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 that 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 7 illustrates this choice. A concrete example follows:

$$\begin{array}{lcll} p_1 & = & (\ 200.0, & 49.200000000000003) \\ p_2 & = & (\ 100.0, & 49.600000000000001) \\ p_3 & = & (-233.33333333333334, & 50.93333333333333\ ) \\ p_4 & = & (\ 166.66666666666669, & 49.333333333333336) \end{array} \qquad \begin{array}{l} \textit{float\_orient}(p_1, p_2, p_3) > 0 \\ \textit{float\_orient}(p_1, p_2, p_4) < 0 \\ \textit{float\_orient}(p_2, p_3, p_4) < 0 \\ \textit{float\_orient}(p_3, p_1, p_4) < 0\ (??) \end{array}$$

The first three points form a counter-clockwise oriented triangle and according to *float_orient*, the algorithm believes 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 of L.

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



(a) **(b)**

Figure 8: Visualization of the region of interest for the points p_1 and p_2 for the Failure B_2 data set. **(a)** Candidates can be chosen from the red regions and must be below the black line. **(b)** Not all candidates will give rise to a proper convex hull for the first four points. All invalid candidates are masked out in light grey.

$$
\begin{array}{lcll}
p_1 & = & (\ 0.50000000000001243,\ 0.50000000000000189) & \textit{float\_orient}(p_1,p_4,p_5) < 0\ (??)\\
p_2 & = & (\ 0.50000000000001243,\ 0.50000000000000333) & \textit{float\_orient}(p_4,p_3,p_5) > 0\\
p_3 & = & (24.00000000000005,\ \ 24.000000000000053\ ) & \textit{float\_orient}(p_3,p_2,p_5) < 0\\
p_4 & = & (24.000000000000068,\ 24.000000000000071\ ) & \textit{float\_orient}(p_2,p_1,p_5) > 0\\
p_5 & = & (17.300000000000001,\ 17.300000000000001\ ) &
\end{array}
$$

Inserting the first four points results in the convex quadrilateral (p_1, p_4, p_3, p_2); this is correct. 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.

We next discuss how we found the instance illustrating Failure B_2. Intuition told us that an example (if it exists) would consist of a quadrilateral with two nearly parallel sides and the two other sides being very short. A query point sitting above the middle of one of the long sides might be able to "float-see" the opposite side of the quadrilateral. It would not see the two short sides. We took the points in Figure 6(a) as a starting point, denote them q_1, q_2, ... We set $p_3 = q_3$, $p_4 = q_2$, $p_5 = q_1$, and decided to look for p_1 and p_2 in the vicinity of q_4. So we searched for points p near q_4 with $\textit{float\_orient}(p, p_4, p_5) < 0$ and $\textit{float\_orient}(p_3, p_1, p) < 0$ that are also below the exact lines defined by (p_3, p_5) and (p_4, p_5) (the last condition ensures that p_5 lies above the quadrilateral). Figure 8(a) visualizes the region of interest for p.

In addition, the first four points should realize a convex hull with our algorithm. In particular, unwanted classifications from *float_orient* as collinear need to be avoided. We mask all forbidden regions in the visualization and we obtain Figure 8(b), from which we were able to select our example points. We selected two points on one of the vertical red lines and below the black line.

Finally, we visualize the region around p_5 in Figure 9. The error is small for *float_orient* in this region, but nevertheless there are several points realizing Failure B_2, of which two are shown in the

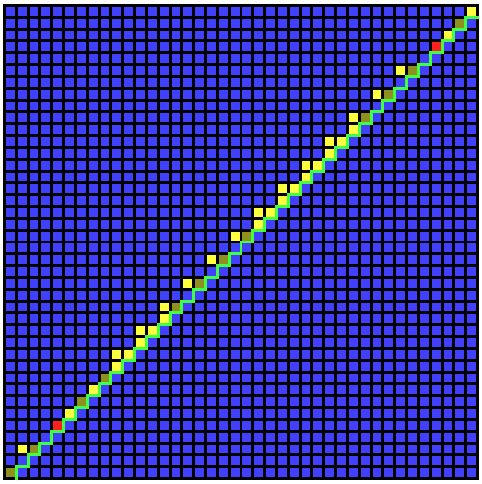


Figure 9: Closeup of the neighborhood of the fifth point that causes Failure B_2, it is the lower left one of the two red pixels, but the other at grid-distance $(32, 32)$ from the first also leads to failure and there are several more candidates not shown in this limited view.

magnified view.

Further Examples: Besides the four logical possibilities above, we can look at quantitative versions:

1. The point sees only a subset of the edges visible to it. Then too few points will be deleted from L.
2. The point sees a superset of the edges visible to it. Then too many points will be deleted from L.

4.3 Global Effects of Failures

By now, we have seen examples that invalidate the correctness properties of the incremental algorithm and we have seen the effect of an incorrect orientation test for a single update step. We next study global effects. *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 towards infinity. The true convex hull has four extreme points. The algorithm misses p_4.

$$
\begin{array}{lcll}
p_1 & = & (0.10000000000000001, 0.10000000000000001) & \mathit{float\_orient}(p_1, p_2, p_3) < 0 \\
p_2 & = & (0.20000000000000001, 0.20000000000000004) & \mathit{float\_orient}(p_1, p_2, p_4) = 0 \text{ (??)} \\
p_3 & = & (0.79999999999999993, 0.80000000000000004) & \mathit{float\_orient}(p_2, p_3, p_4) = 0 \text{ (??)} \\
p_4 & = & (1.267650600228229 \cdot 10^{30}, 1.2676506002282291 \cdot 10^{30}) & \mathit{float\_orient}(p_3, p_1, p_4) > 0
\end{array}
$$

The algorithm crashes or does not terminate: See Failure B_1.

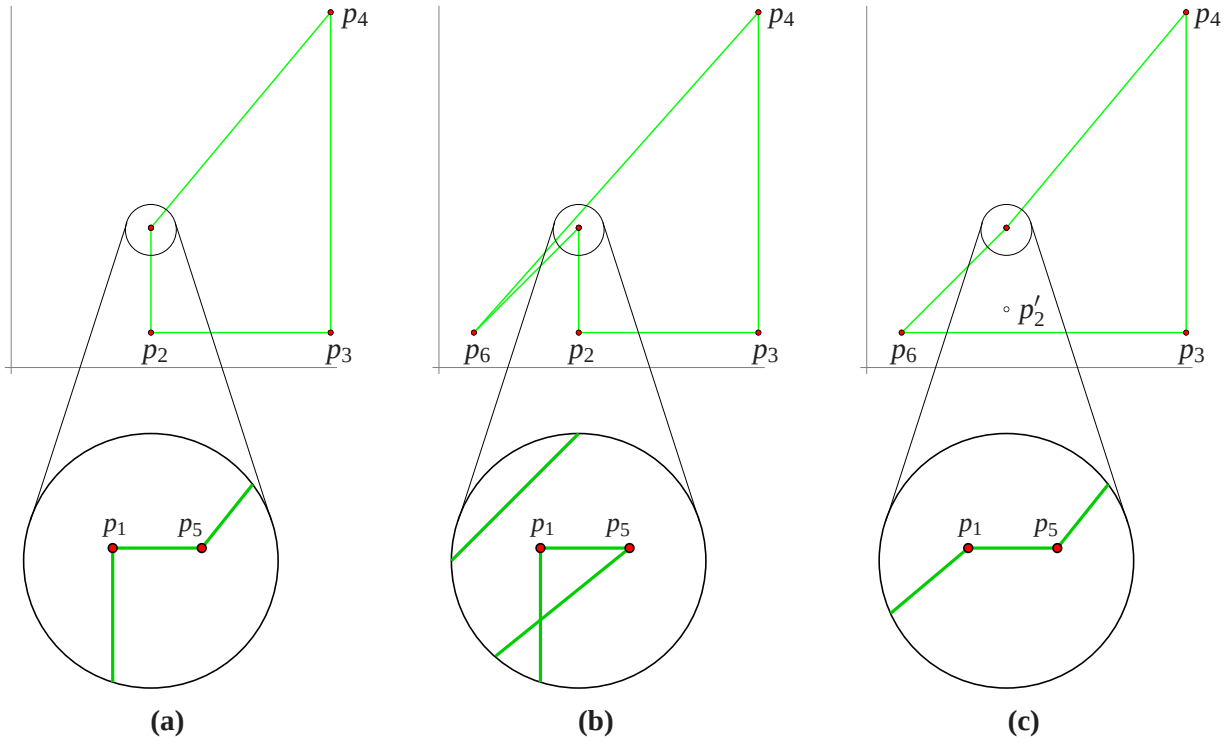


Figure 10: **(a)** The hull constructed after processing points p_1 to p_5. Points p_1 and p_5 lie close to each other and are indistinguishable in the upper figure. The magnified schematic view below shows that we have a concave corner at p_5. The point p_6 sees the edges (p_1, p_2) and (p_4, p_5), but does **not** see the edge (p_5, p_1). One of the former edges will be chosen by the algorithm as the chain of edges visible from p_6. Depending on the choice, we obtain the hulls shown in **(b)** or **(c)**. In **(b)**, (p_4, p_5) is found as the visible edge, and in **(c)**, (p_1, p_2) is found. We refer the reader to the text for further explanations. The figures show the coordinate axes to give the reader a frame of reference.

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:

$$
\begin{array}{lcl}
p_1 & = & (24.00000000000005,\ \ 24.000000000000053) \\
p_2 & = & (24.0,\ \ 6.0\ ) \\
p_3 & = & (54.85,\ \ 6.0\ ) \\
p_4 & = & (54.850000000000357,\ \ 61.000000000000121) \\
p_5 & = & (24.000000000000068,\ \ 24.000000000000071) \\
p_6 & = & (\ 6.0,\ \ 6.0\ ).
\end{array}
$$

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. Point p_5 lies inside the convex hull of the first four points; but *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 10(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, namely (b). This is because in our sample implementation as given in the appendix, we have $L = (p_2, p_3, p_4, p_1)$, and hence the search for a visible edge starts at edge (p_2, p_3). In order to produce (c) with our implementation we replace the point p_2 by the point $p_2' = (24.0, 10.0)$. Then p_6 sees (p_2', p_3) and identifies (p_1, p_2', p_3) as the chain of visible edges and hence constructs (c).

5 Incremental 3D Delaunay Triangulation Algorithm

The planar convex hull algorithms are simple educational examples. A more complex, and in practice quite relevant algorithm is the incremental construction of the 3d Delaunay triangulation, such as the one found in CGAL. The complex algorithm consists of several phases, which all can fail in different ways when executed with floating-point arithmetic. We describe the algorithm in more detail and give a numerical example that causes an infinite loop. It is not easy to hand-construct such an example, but we provide an algorithm that easily finds many such examples.

We say that a point u is *in conflict* with a tetrahedron t if u lies in the interior of the circumscribing sphere of t. A Delaunay triangulation of a set of points is a triangulation in which all tetrahedra verify the *Delaunay property*: they do not conflict with any other point of the triangulation. In the degenerate case of co-spherical points, the Delaunay triangulation may not be unique.

The incremental Delaunay algorithm inserts a new point u in the current Delaunay triangulation in two steps: point location and update. The point location step returns a tetrahedron in conflict with u. The update step removes all tetrahedra in conflict with u and populates the resulting hole with new tetrahedra connecting u with the facets of the hole, thus establishing the Delaunay property for the resulting triangulation.

One way to implement the point location step is to find a tetrahedron that contains u (there can be several in the case that u is on a facet or an edge), which will a fortiori be in conflict with u. Several algorithms can be used here, but we focus on a specific walking algorithm called *remembering stochastic walk* in [DPT02], which traverses the adjacency relations between tetrahedra. The walking part is usually sped up by another algorithm that quickly finds a tetrahedron near the target, using, for example, either a hierarchy of triangulations or a small random sample of the points. However, we concentrate in this paper on studying the robustness of the fundamental walking part.

Note that inserting a point that is outside the convex hull of the existing triangulation can be performed similarly but uses different predicates. We are not studying the failures that can be found in such cases. So in the sequel, we assume that u lies inside the convex hull of the previous points. Furthermore, we do not consider the first phase of the incremental construction algorithm where an initial full-dimensional triangulation is constructed, because this phase requires additional predicates.

5.1 Failures of the Point Location Step

By convention and ensured by the algorithm, all tetrahedra in the triangulation are positively oriented, i.e., $\mathit{orientation}(p, q, r, s)$ is positive, where the three-dimensional orientation test is defined analogously to the planar orientation test in Equation 1 as:

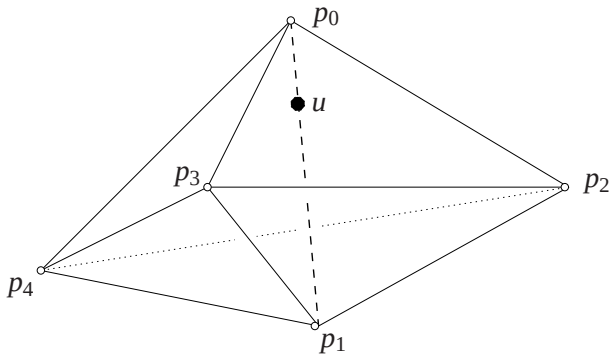


Figure 11: Inserting a point u near the central edge (p_0, p_1) of a Delaunay triangulation made of three tetrahedra around that central edge.

$$orientation(p,q,r,s) = sign(\det \begin{bmatrix} 1 & p_x & p_y & p_z \\ 1 & q_x & q_y & q_z \\ 1 & r_x & r_y & r_z \\ 1 & s_x & s_y & s_z \end{bmatrix}). \tag{3}$$

The facet (q,r,s) of a tetrahedron opposite to p is said to *separate* the tetrahedron from a point u if $orientation(u,q,r,s)$ is negative. The definition extends analogously to the facets opposite of q, r, and s, respectively, replacing the point opposing the facet with u in the orientation predicate.

The point location algorithm starts at an initial tetrahedron (p,q,r,s), iterates over its four facets, and tests if a facet separates the tetrahedron from u. If such a facet is found, the algorithm moves to the neighbor tetrahedron and repeats the point location. Otherwise, no such facet is found and u is inside or on the boundary of the tetrahedron, which means that u is either in conflict or is equal to one of the vertices of the tetrahedron. The latter case can be seen immediately from the return values of all the orientation predicates performed with the facets of the tetrahedron and u.

Property E. The point location algorithm terminates with a tetrahedron that contains the query point u if the triangulation fulfills the Delaunay property and u is inside the convex hull of the triangulation [Ede90].

If we use a corresponding floating-point implementation for our orientation test we observe that Property E can fail in two ways: (1) the algorithm does not terminate, or (2) the algorithm returns a tetrahedron that does not contain u (but which may still be in conflict with u and thus not endangers the update step). We confine ourselves to the first kind of failure:

Failure E_1: The point location algorithm does not terminate. The termination proof relies on the acyclicity property of the Delaunay triangulation and the correct evaluation of the orientation predicate. We search for a cycle among a small number of tetrahedra. Two tetrahedra are actually not enough because of the obvious optimization that the algorithm does not test the tetrahedron again where it came from. Three tetrahedra may suffice, where u lies close to the three supporting planes of the three common facets to trigger numerical inaccuracies in the orientation test. This suggests to build a triangulation with a central edge surrounded by three tetrahedra and to locate a point u that is approximately on this edge as illustrated in Figure 11.

We provide a program that creates random examples of that nature and tests them for Failure E_1. At first, the program generates five random points and verifies that their Delaunay triangulation has

the desired shape of three tetrahedra grouped around a central edge, and if not it tries another set of points. Then, the program generates a point u near the central edge by computing a point on the edge using approximate floating-point computations. At the end, the program locates u in the triangulation. In fact, the point location does not terminate quite often due to inconsistent answers of the orientation predicate in the volume around the edge. We give here an example data set that sends the algorithm into an infinite loop:

$$
\begin{aligned}
p_0 &= (0.092408271079090554,\ 0.1326565794620080400,\ 0.20816329990430305)\\
p_1 &= (0.183729934258721530,\ 0.0085360395142579648,\ 0.39535821959993456)\\
p_2 &= (0.382775030788625510,\ 0.2050904804319415600,\ 0.01038994374388480)\\
p_3 &= (0.256251824311654270,\ 0.6315717178093045400,\ 0.16190908040221075)\\
p_4 &= (0.191845325127811610,\ 0.0281530165464261020,\ 0.57432720440646179)\\
\\
u &= (0.232038626475695340,\ 0.4235560948517673200,\ 0.23985175657768110)
\end{aligned}
$$

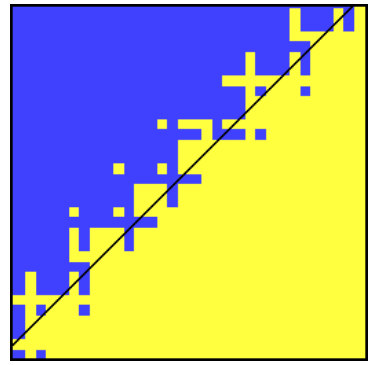
6 Non-Solutions

A number of approaches have been suggested to make floating point implementations work, either of specific algorithms or in general. We point to promising approaches in the next section and discuss two frequently suggested approaches that do not work in this section.

The first approach is specific to the planar convex hull problem. A frequently heard reaction to our paper is that all our examples exploit the fact that the first few points are nearly collinear. If one starts with a "roundish" hull, or at least starts with a hull formed from the points of minimal and maximal x- and y- coordinates, the problem will go away. We have two answers to this suggestion: Firstly, neither way can cope with the situation that all input points are nearly collinear, and secondly, the example in Figure 10 falsifies this suggestion. Observe that we have a "roundish" hull after the insertion of the points p_1 to p_4 and then the next two insertions lead the algorithm astray. The example can be modified to start with points of minimal and maximal x- coordinates first, which we suggest as a possible course exercise.

Epsilon-tweaking is another frequently suggested and used remedy, i.e., instead of comparing exactly with zero, one compares with a small (absolute or relative) tolerance value epsilon. Epsilon-tweaking simply activates rounding to zero. In the planar hull example, this will make it more likely for points outside the current hull not to see any edges because of enforced collinearity and hence at least failure A_1 will still occur. In our examples of Section 3, the yellow band in our visualizations of collinear pixels becomes wider, but its boundary remains as fractured as it is in the comparison with zero, see Figure 12.

Another objection argues that our examples are unrealistic since they contain near collinear point triples or points very close together (actually the usual motivation for Epsilon-tweaking). Of course, the examples have to look like this, otherwise there would not be room for rounding errors. But they are realistic; firstly, practical experience shows it. Secondly, degeneracies, such as collinear point triples, are on purpose in many data sets, since they reflect the design intent of a CAD construction or in architecture. Representing such collinear point triples in double precision arithmetic and further transformations lead to rounding errors that turn these triples into close to collinear point triples. And thirdly, increasingly larger data sets increase the chance to have a bad triple of points just by bad luck, and a single failure suffices to ruin the computation.

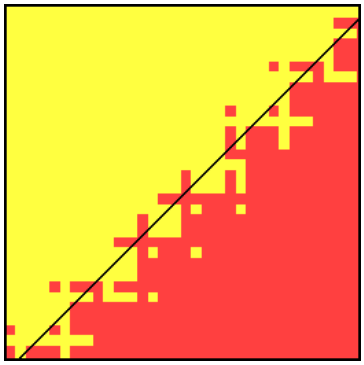


p : $\begin{pmatrix} 0.5 \\ 0.50000000000833222 \end{pmatrix}$

q : $\begin{pmatrix} 12 \\ 12 \end{pmatrix}$

r : $\begin{pmatrix} 24 \\ 24 \end{pmatrix}$

(a)



p : $\begin{pmatrix} 0.50000000000833222 \\ 0.5 \end{pmatrix}$

q : $\begin{pmatrix} 12 \\ 12 \end{pmatrix}$

r : $\begin{pmatrix} 24 \\ 24 \end{pmatrix}$

(b)

Figure 12: The effect of epsilon-tweaking: The figures show the result of repeating the experiment of Figure 2(a), but using an absolute epsilon tolerance value of $\varepsilon = 10^{-10}$, i.e., three points are declared collinear if *float_orient* returns a value less than or equal to 10^{-10} in absolute value. The yellow region of collinearity widens, but its boundary is as fractured as before. Figure (a) shows the boundary in the direction of the positive y-axis, and Figure (b) shows the boundary in the direction of the positive x-axis. The figures are color coded: Yellow (red, blue, resp.) pixels represent collinear (negative, positive, resp.) orientation. The black lines correspond to the lines $\mathit{orientation}(p,q,r) = \pm\varepsilon$.

7 Conclusion

We provided instances that 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 think 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.

We want to reiterate that our goal was not to show that the specific algorithms discussed in this paper can fail, but to give illustrative examples for what can go wrong and why. We could have used other algorithms and implementations as the starting point of our work. After all, it is well-known that most geometric algorithms fail for some inputs, if implemented with floating-point arithmetic naïvely. We have chosen the specific algorithms because they are frequently taught and because they are so simple that one can actually discuss in full detail what goes wrong. In particular, in the incremental convex hull algorithm, we kept the search for a first visible edge as simple as possible. After all, it is less important how an initial visible edge is found. It is only important which of the inspected edges are declared visible by *float_orient*. Thus, with randomized incremental algorithms, that use more sophisticated strategies to search for an initial visible edge, we would get the same kind of failures. Moreover, this is not a study on the numerical stability of planar convex hull algorithms. We see our contribution in presenting educational examples for the bigger problem of why and how geometric algorithms can fail, studied on a level where all aspects of the problem can still be discussed and

understood in class. We hope that the examples will raise awareness for the problem and willingness to study the various approaches to reliable geometric computation.

We do not want to leave our readers in despair and therefore close with some pointers to successful approaches to reliable geometric computation. There are several approaches: (1) make sure that the implementations of geometric predicates always returns the correct result or (2) change the algorithm so that it can cope with the floating-point implementation of its geometric predicates 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 [HS98, FKMS05].

The first approach, known as the exact geometric computation (EGC) paradigm, has been adopted for the software libraries LEDA, CGAL and CORE LIBRARY [KN04, FGK+00, MN99, KLPY99]. In the second approach the interpretation of "meaningful" is a crucial and difficult problem. For convex hull and Delaunay triangulations there are more robust algorithms [BDH96, DSB92, For89, For95, GSS90, JW94, JS06, KW98, LM90]. For further references to these approaches we refer the reader to [Yap04, Sch00].

References

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

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[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.

[DH91] P. Deuflhard and A. Hohmann. *Numerische Mathematik: Eine algorithmisch orientierte Einführung*. Walter de Gruyter, 1991.

[DPT02] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

[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.

[Ede90] H. Edelsbrunner. An acyclicity theorem for cell complexes in *d* dimensions. *Combinatorica*, 10(3):251–260, 1990.

[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.

[FKMS05] S. Funke, Ch. Klein, K. Mehlhorn, and S. Schmitt. Controlled perturbation for Delaunay triangulations. In *Proc. of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1047–1056, Vancouver, Canada, 2005.

[For70] G. E. Forsythe. Pitfalls in computation, or why a math book is not enough. Technical Report CS-TR-70-147, Stanford University, Computer Science Department, 1970. available at `historical.ncstrl.org/litesite-data/stan/CS-TR-70-147.pdf`.

[For85] A. R. Forrest. Computational geometry in practice. In R. A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, volume F17 of *NATO ASI*, pages 707–724. Springer-Verlag, 1985.

[For89] S. Fortune. Stable maintenance of point-set triangulations in two dimensions. *Proc. Foundations of Computer Science (FOCS)*, 30:494–499, 1989.

[For90] S. Fortune. Stable maintenance of point-set triangulation in two dimensions. manuscript, 1990.

[For95] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Internat. J. Comput. Geom. Appl.*, 5(1–2):193–213, 1995.

[FvW96] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15:223–248, 1996. preliminary version in ACM Conference on Computational Geometry 1993.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

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

[GSS90] L. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. In *Proc. 1st Annu. SIGAL Internat. Sympos. Algorithms*, volume 450 of *Lecture Notes Comput. Sci.*, pages 261–270. Springer-Verlag, 1990.

[HS98] D. Halperin and C.R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comp. Geom.: Theory and Applications*, 10, 1998.

[IEE87] IEEE standard for binary floating-point arithmetic. *SIGPLAN Notices*, 22(2):9–25, February 1987. Reprint of ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA.

[JS06] D. Jiang and N. F. Stewart. Backward error analysis in computational geometry. In *ICCSA (1)*, pages 50–59, 2006.

[JW94] J. W. Jaromczyk and G. W. Wasilkowski. Computing convex hull in a floating point arithmetic. *CGTA: Computational Geometry: Theory and Applications*, 4, 1994.

[KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[KN04] L. Kettner and S. Näher. Two computational geometry libraries: LEDA and CGAL. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 65, pages 1435–1463. CRC Press LLC, Boca Raton, FL, second edition, 2004.

[KW98] Lutz Kettner and Emo Welzl. One sided error predicates in geometric computing. In *IFIP World Computer Congress on Fundamentals - Foundations of Computer Science*, pages 13–26. Österreichische Computer Gesellschaft, 1998.

[LM90] Z. Li and V.J. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In *SoCG'90*, pages 235–243. ACM Press, 1990.

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

[Sch00] S. Schirra. Robustness and precision issues in geometric computation. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

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

[ST06] S. Schirra and J. Tusch. Experimental comparison of the cost of approximate and exact convex hull computation in the plane. In *Proceedings of the 18th Canadian Conference on Computational Geometry (CCCG'06)*, pages 19–22, 2006.

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

[Yap04] 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, 2004.

Appendix: Implementation of the Incremental Algorithm

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 companion web page `http://www.mpi-inf.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 left turn, it returns zero if they are collinear, and it returns -1 if they form a right turn. 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 it 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 `result` output iterator at the end of the function. It also returns the modified `result` iterator.

```
template <typename ForwardIter, typename OutputIter>
OutputIter incr_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);
}
```