Appeared in Proc. *15th IFIP World Computer Congress, Fundamentals – Foundations of Computer Science*,
Kurt Mehlhorn (Eds.), pp. 13-26, August 1998.

One Sided Error Predicates in Geometric Computing

Lutz Kettner and Emo Welzl
Inst. of Theoretical Computer Science
ETH Zürich*

Abstract

A *conservative* implementation of a predicate returns true only if the *exact* predicate is true. That is, we accept a one sided error for the implementation. For geometric predicates, such as orientation- or incircle-tests, this allows efficient floating point implementations of the predicates with rare occurrences of the one sided error.

We discuss the use of such conservative implementations for convex hull and triangulation algorithms for point sets in the plane. The resulting programs show a minor slowdown compared to an implementation that completely ignores the finite precision issue. However, our programs always produce output that satisfies basic desirable properties. The output can be easily checked for correctness and – if necessary – it can be repaired with an exact implementation of the needed predicates.

Although (or since?) conservative implementations of predicates cannot detect degeneracies, the programs work for degenerate input. In fact, in our experiments the advantage in running time compared to exact implementations of predicates (based on floating point filters) was most apparent for highly degenerate inputs.

1 Introduction

The field of Computational Geometry has produced a rich body of algorithms for geometric problems. A number of fundamental techniques have been designed, and key problems and problem classes have emerged; see the textbooks [Meh84, PS85, Ede87, ORo94, Mul94, Kle97, Ber*97, BY98] and the handbooks [GO97, SU98]. To a large extent the theory has been developed with asymptotic complexity analysis and under the assumption of the real RAM model, where computations with real numbers can be performed in constant time. For most algorithms and problems this is a justified assumption that can be perfectly simulated with finite precision numbers, if the input has limited precision[1]. However, when geometric algorithms are actually transformed into programs unpleasant effects can occur. We might ignore the finite precision issue, i.e., arithmetic operations on floating

*Support from the ESPRIT IV LTR Project No. 21957 (CGAL) is acknowledged.

[1]Clearly, that is questionable, since the larger the input, the larger the numbers we need. This, however, is not what we are concerned about here.

point numbers give only approximate results. Then we often experience that sophisticated algorithms become very sensitive to numerical problems, more than many brute force methods. The lack of accuracy is not our primary concern here. Rather, the numerical problems may destroy geometric consistency an algorithm may rely on[2]. As a result the program may crash, may run into an infinite loop, or – perhaps worst of all – may produce unpredictable erroneous output.

Hence, this is a crucial issue in almost every effort for implementing geometric algorithms, and for libraries of geometric programs in particular. Examples are the geometry part of the Library of Efficient Data types and Algorithms LEDA[3] [MN95, Meh*97], the Computational Geometry Algorithms Library CGAL[4] [Fab*98], the GEOMLIB[5] project [Bak*97], or the rich collection of geometric programs in the Directory of Computational Geometry Software[6] [Ame97].

As mentioned before, the numerical problems can usually be circumvented by simulating infinite precision. Several successful techniques have recently been proposed in order to keep the slowdown small – see [Bur*95, FW96, She97, Avn*97] – , but some penalty in running time has to be accepted. This is particularly annoying, since a brute-force implementation may encounter wrong answers only rarely – but it may!

Instead of using the more expensive exact implementation of the predicate throughout the algorithm, we suggest to start with a simple and fast implementation of the predicate that is allowed to have a one-sided error. On the one hand, this error occurs rarely as experience shows, see e.g. [DP98] for an analysis of random points. On the other hand, we show that there are algorithms that can be implemented using this simple implementation such that the generated output still guarantees a number of desirable properties. For example the correctness of the output can be easily checked and repaired with an exact implementation of the predicate, if this is necessary.

Predicates. We consider points p in the plane given by their cartesian coordinates $p = (p_x, p_y)$. The geometric predicates which we will need here are the *lexicographic order* for points p and q

$$p < q \;\; :\Longleftrightarrow \;\; p_x < q_x \;\text{ or }\; (p_x = q_x \;\text{ and }\; p_y < q_y)\,,$$

and the *sidedness-* (or orientation-) *predicate* for points p, q, and r

$$p \text{ left of } qr \;\; :\Longleftrightarrow \;\; \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} > 0\;.$$

In geometric terms, p is left of qr, iff q and r are distinct points, and p lies left of the directed line through q and r that encounters q and r in this order. There are several ways to evaluate this determinant. A possible implementation is[7]

```
return (q.x-p.x)*(r.y-p.y) - (q.y-p.y)*(r.x-p.x) > 0;
```

[2]Imagine an algorithm learns for variables a, b and c that $a < b$, $b < c$, and $c < a$, which contradicts transitivity.
[3]`http://www.mpi-sb.mpg.de/LEDA/leda.html`
[4]`http://www.cs.ruu.nl/CGAL/`
[5]`http://www.cs.jhu.edu/labs/cgc/`
[6]`http://www.geom.umn.edu/software/cglist/`
[7]We present program examples in C++ syntax. Note the conventional correspondences, e.g. `p[i]` and p_i or `p.x` and p_x.

Clearly, rounding errors may lead to wrong results if we use floating point arithmetic to evaluate the expression. From such an implementation, we should not and cannot expect even the basic geometric consistency that p left of qr iff r left of pq iff q left of rp. For example, about one to three inconsistent triples occurred among the $\binom{1000}{3}$ triples from 1000 random points in $[0,1) \times [0,1)$ in an experiment using IEEE single precision coordinates. However, no such inconsistencies were observed for IEEE double precision (although they will happen).

An implementation of a predicate is *conservative* if it returns true only if the considered predicate is true, and we will call such an implementation a *conservative predicate*, for short. We describe possible implementations of left-of predicates, which are conservative under certain assumptions on the input coordinates and the use of the IEEE 754 arithmetic standard [IEEE].

First, let us assume that the input coordinates lie in the range $[B, 2B]$ for some number B. Then – according to Sterbenz' Theorem [Ste74] (cf. [Hig96, Theorem 2.5]) – the difference between two such numbers is evaluated exactly in IEEE floating point arithmetic. Next, if an expression of the form

```
a*b - c*d
```

is positive (evaluated in floating point arithmetic using round-to-nearest), then the underlying exact value $ab - cd$ is positive. Still, the rounded expression may evaluate to 0 because of rounding errors in the multiplication. In any case, the ad hoc implementation as presented above is conservative under the given assumption.

The typical approach for floating point filters is to derive a threshold t such that if the absolute value of the predicate expression is greater than t the sign of the expression is known to be correct. Otherwise an exact implementation (or refined filter with better threshold) is needed to determine the sign. If we compare the expression itself instead of its absolute value with the threshold, we gain a conservative predicate. We save the absolute value computation, which might involve a function call or a two-sided test with an additional branching instruction, though it could be as simple as to set a single bit. Also, the conservative predicate does not retract to an exact implementation, which can be superfluous in an algorithm that is able to decide later that this decision is not needed at all. An example are convex hulls, where there is no need for a degenerate point set to bother the algorithm if the algorithm will detect anyway that this point set is inside the final convex hull.

The threshold can be computed at runtime, e.g. depending on the actual parameter sizes [Bur*95, She97], or it can be based on a static analysis with a priori assumptions on the parameter sizes [FW96]. If the input coordinates lie in the range $[-B, +B]$ for some positive number B, a static threshold $t = 8(3u + 6u^2 + 4u^3 + u^4)B^2$ can be derived using standard error analysis (e.g. along the lines of [She97, page 348]), where u is the unit roundoff of the number system. We have $u = 2^{-53}$ for IEEE double precision, and $u = 2^{-24}$ for IEEE single precision floating point numbers. Note that an actual implementation will need a threshold $t' \geq t$ that is representable in the floating point number system. The error analysis that yields this threshold is generally applicable, while the use of Sterbenz' Theorem is limited to simple predicates.

By now the reader may wonder, what all the effort is for. There is always an easy way for a conservative implementation: just return false. However, our point is that the conservative predicates as suggested will rarely give the wrong answer. For example, in all our experiments[8] we encountered not a single instance of three random points with *double* precision coordinates in $[0,1) \times [0,1)$ where the

[8] This experimental evidence is substantiated by results in [DP98].

absolute value of the sidedness determinant was smaller than the threshold given above. Such cases had only been observed with input sets of 10^6 random points "on" the unit circle[9].

Here an immediate objection may be: "Inputs are not random, they often contain (intended) degeneracies, where the sidedness determinant vanishes and lies between the thresholds." But note, a conservative sidedness-predicate is never wrong on a degeneracy, although it cannot verify it! In fact, the benefit of using conservative predicates is most apparent for highly degenerate input, as already indicated above.

In the remaining part we describe two known algorithms for convex hull computation from the literature. We briefly recapitulate their description, with careful attention to details in order to make their applicability for conservative predicates explicit. Then we report on experimental results of the implementation of the algorithms.

Notation. Input for programs comes usually as sequences not as sets, which sometimes entails the simplest degeneracy: repeated copies of the same point. We emphasize this distinction and use $P, Q, \ldots$ for point sequences, and $\tilde{P}, \tilde{Q}, \ldots$ for the sets of points occurring in those sequences. If we consider sets to begin with, we use calligraphic letters $\mathcal{P}, \mathcal{Q}, \ldots$.

2 Two convex hull algorithms

We describe two algorithms for the convex hull computation in the plane that are suitable for conservative sidedness predicates: *successive local repair* (SLR) – a variant of Graham's scan [Gra72, For89], and *randomized incremental construction* (RIC) with history graph [CS89, BT86]. Both algorithms compute separately the lower and the upper chain of a polygon bounding the convex hull. The descriptions are restricted to the lower hull.

Lower convex hull polygons. A sequence $Q = (q_0, \ldots, q_{k-1})$ of k points is *lex-increasing*, if $q_{i-1} < q_i$ for all $1 \leq i < k$. Q is a *lower convex hull (LCH-) polygon* of a point set $\mathcal{P}$, if (i) $\tilde{Q} \subseteq \mathcal{P}$, (ii) $q_0 = p_{\text{min}}$, the lexicographically smallest point in $\mathcal{P}$, and $q_{k-1} = p_{\text{max}}$, the lexicographically greatest point in $\mathcal{P}$, and (iii) the polygon determined by Q traces the lower part of the convex hull of $\mathcal{P}$ from p_{min} to p_{max}. Q is called *an LCH-conserving polygon* of $\mathcal{P}$, if (i) $\tilde{Q} \subseteq \mathcal{P}$, (ii) Q is lex-increasing, and (iii) there is a subsequence $(q_{i_0}, q_{i_1}, \ldots, q_{i_{l-1}})$, $i_0 < i_1 < \ldots < i_{l-1}$, of Q that is an LCH-polygon of $\mathcal{P}$.

An LCH-polygon must contain all vertices of the lower convex hull of $\mathcal{P}$, but it may also contain other points from the boundary of the convex hull. An example of an LCH-conserving polygon of $\mathcal{P}$ is the lex-increasing sequence of all points in $\mathcal{P}$. Note that lex-increasing does not allow repetitions of points in the sequence.

Observation 1 *Let $\mathcal{P}$ be a point set. If $p, p', p'' \in \mathcal{P}$ satisfy*[10] *$p' \leq p \leq p''$ and p* left of *$p'p''$, then p appears in no LCH-polygon of $\mathcal{P}$.* 

[9] We put "on" in quotation marks, since the generated points are not exactly on a circle due to rounding.

[10] We refer here to the lexicographic ordering: $p \leq q \iff p < q$ or $p = q$.

Observation 2 *If $Q = (q_0, \ldots, q_{k-1})$ is an LCH-conserving polygon of a point set $\mathcal{P}$, and for* no $i, 1 \leq i < k-1$, *holds q_i left of $q_{i-1}q_{i+1}$, then Q is an LCH-polygon.* 

Successive local repair. Observations 1 and 2 suggest a simple scheme for computing an LCH-polygon of a point sequence P. We start with the lex-increasing sequence Q of the points in $\tilde{P}$, i.e. we have to sort P and to remove repetitions. Whenever we find a consecutive triple (p, q, r) in Q with q left of pr, we remove q from the sequence. Observation 1 guarantees that the sequence remains LCH-conserving. As soon as no such triple exists, Observation 2 ensures that the resulting sequence is an LCH-polygon of $\tilde{P}$.

We call this scheme *successive local repair (SLR)*. For an implementation, it remains to specify in which order triples are visited. The reader is encouraged to recognize Graham's scan or Divide-and-Conquer as variants of this scheme depending on the order chosen. We suggest to proceed as in Graham's scan.

```
// p[0..n-1] lex-increasing sequence of n > 0 points.
q[0] = p[0];
int k = 1;
for (int i = 0; i < n; i = i + 1) {
    while ( k > 1 && q[k-1].left_of(q[k-2], p[i]))
        k = k - 1;
    q[k] = p[i];
    k = k + 1;
}
// q[0..k-1] is an LCH-polygon of p[0..n-1].
```

$O(n \log n)$ time is needed to sort the points in P and to remove repeated points. A simple analysis shows that the procedure from above is linear in n. It uses exactly $n - k$ left-of tests which return true and at most $n - 2$ left-of tests which return false.

If we use a conservative implementation of the left-of predicate with the SLR scheme, we still gain the following useful properties for the output:

- *Observation* 1 *guarantees that we obtain an LCH-conserving polygon of the input points.*
- *The result $Q = (q_0, \ldots, q_{k-1})$ has all points of $\tilde{P}$ on or above it.*
- *The SLR-scheme can be applied to $Q = (q_0, \ldots, q_{k-1})$ a second time with an exact left-of predicate, which can check whether the sequence is already an LCH-polygon of $\tilde{P}$, or can perform further local repairs, otherwise.*

Triangulation. SLR can also be used to compute a triangulation of a point set $\mathcal{P}$. A triangulation of $\mathcal{P}$ is a collection of closed triangles whose union covers the convex hull of $\mathcal{P}$ and whose interiors are pairwise disjoint. The vertices of the triangles are points from $\mathcal{P}$, and no other part of the triangles contains any point from $\mathcal{P}$.

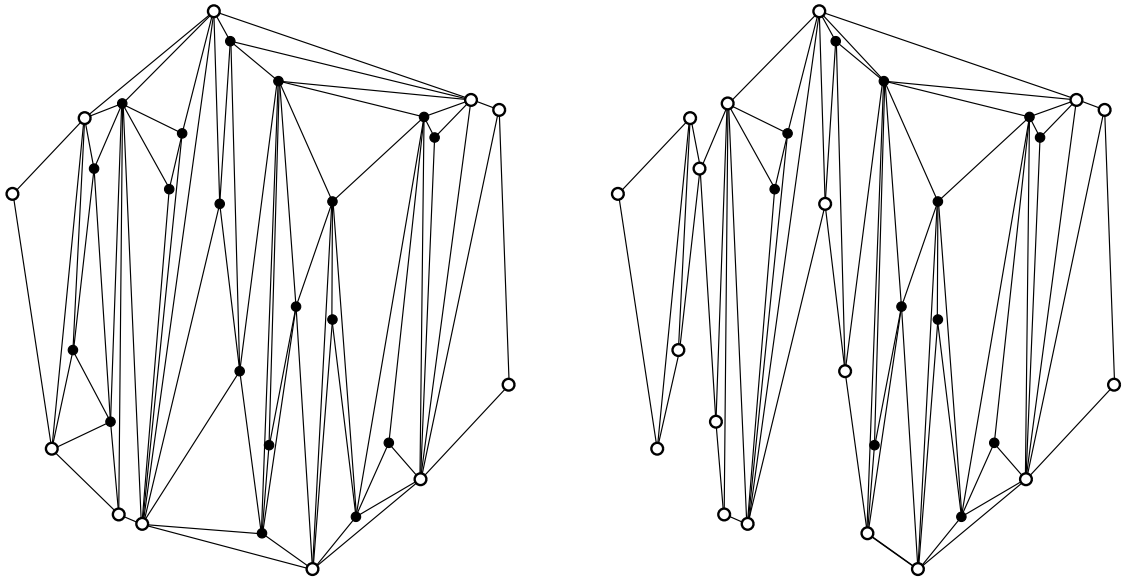

Figure 1: Two triangulations obtained by the SLR algorithm. For the triangulation to the left a conservative implementation of the sidedness predicate with threshold was used. For the triangulation to the right, the same predicate was switched from true to false in 30% of the calls at random. It represents the same triangulation as the one to the left minus the missing triangles at the boundary and with a single diagonal swapped in the upper right part.

If we generate a triangle with vertices (p_i, q_{k-1}, q_{k-2}) whenever q_{k-1} is left of $q_{k-2}p_i$, the algorithm from above produces such a triangulation. Even if we run the algorithm with a conservative left-of predicate we keep all properties of a triangulation, except that the union of all triangles does not necessarily cover the convex hull of the input points. In particular, we can guarantee that all triangles have disjoint interiors, they are non degenerate, and their vertices (p_i, q_{k-1}, q_{k-2}) are in counterclockwise order.

Randomized incremental construction. The $O(n \log n)$ time bound of SLR is worst-case optimal, but there are better solutions available if the number of vertices of the convex hull is small. As an example we consider a randomized incremental construction. This algorithm processes one input point after the other in random order and maintains the LCH-polygon for all points seen so far. A central step in the algorithm is the decision, whether a new point is above the current LCH-polygon and can be discarded, or whether the point needs to be inserted in the LCH-polygon and, if so, where. This *location problem* will be solved with the help of a so called *history graph*, introduced as influence graph in [BT86]. This data-structure records all changes made to the LCH-polygon in the course of the algorithm.

Since we are heading for an algorithm that has an expected running time linear in the size of the input in many cases, we cannot afford to sort the input sequence or to remove point repetitions. In principal, a random permutation of the input points is needed as a preprocessing step to guarantee the expected running time as explained below.

We are given a sequence $P = (p_0, \ldots, p_{n-1})$, $n \geq 1$, of points in the plane with p_{min} the lexicographically smallest and p_{max} the lexicographically largest point in the sequence. For the description, we define $\mathcal{P}_{-1} = \{p_{\text{min}}, p_{\text{max}}\}$ and $\mathcal{P}_i = \mathcal{P}_{i-1} \cup \{p_i\}$, $0 \leq i < n$.

If $p_{\min} = p_{\max}$ we return $(p_{\min})$ as LCH-polygon. Otherwise, we start with $Q_{-1} = (p_{\min}, p_{\max})$, the LCH-polygon of $\mathcal{P}_{-1}$, and successively compute Q_i, the LCH-polygon of $\mathcal{P}_i$. We assume for now that we are given the history graph, which decides for the i-th input point $p = p_i$ whether it is above Q_{i-1} and can be ignored, or it returns the lexicographic position of p in Q_{i-1}, i.e. an index j such that $q_j \leq p \leq q_{j+1}$. If $p = q_j$ or $p = q_{j+1}$, the point p can be ignored and $Q_i = Q_{i-1}$. Otherwise, we insert p between q_j and q_{j+1}. The result is an LCH-preserving polygon of $\mathcal{P}_i$, on which we apply the successive local repair scheme to obtain finally the LCH-polygon Q_i. However, we can restrict the successive local repair to triples that have p as first or as last element.

For the history graph, we maintain the current LCH-polygon Q_i as a doubly-linked list of its edges. That is, for every pair (q_j, q_{j+1}) of adjacent points in Q_i, we have an edge e with a reference $e_{\text{prv\_p}}$ to the point q_j, a reference $e_{\text{nxt\_p}}$ to the point q_{j+1}, a reference $e_{\text{prv\_e}}$ to the previous edge for the pair (q_{j-1}, q_j), and a reference $e_{\text{nxt\_e}}$ to the next edge for the pair (q_{j+1}, q_{j+2}). For the first and last edge the respective dangling references are set to NULL.

The algorithm will keep all edges, even if they disappear from the current polygon Q_i. A boolean flag e_{inner} is used to mark inner edges inherited from previous stages. The flag is false for the edges of the current LCH-polygon. Whenever an edge e disappears from the current polygon, its x-range is covered by one or both of the new edges of the polygon. We will re-use $e_{\text{prv\_e}}$ and $e_{\text{nxt\_e}}$ to reference these new edges. More precisely, let e' and e'' be the new edges such that $e'_{\text{nxt\_e}} = e''$, and let p be their common endpoint: $p = e'_{\text{nxt\_p}} = e''_{\text{prv\_p}}$. If $e_{\text{prv\_p}} \leq p < e_{\text{nxt\_p}}$ we set $e_{\text{prv\_e}} = e'$ and $e_{\text{nxt\_e}} = e''$. If $e_{\text{nxt\_p}} \leq p$ we set $e_{\text{prv\_e}} = e'$ and consider $e_{\text{nxt\_e}}$ as undefined, and for the remaining case of $p < e_{\text{prv\_p}}$ we set $e_{\text{prv\_e}} = e''$ and $e_{\text{nxt\_e}} = e''$. See Figure 2 for an illustration of this data structure after the insertion of several points. Using this history graph, we can now easily search for the location of the new point p in the current LCH-polygon, or detect on the way that p cannot appear in the new LCH-polygon; see the program fragment below. Whenever we make the test p left of $e_{\text{prv\_p}}e_{\text{nxt\_p}}$, the invariant $e_{\text{prv\_p}} \leq p \leq e_{\text{nxt\_p}}$ holds. Thus, if the test returns true, Observation 1 allows us to remove p and proceed to the next point. Note that in fact the stronger invariant $e_{\text{prv\_p}} \leq p < e_{\text{nxt\_p}}$ holds, except where we locate a point $p = p_{\max}$. This is the reason for the asymmetric treatment of the $e_{\text{nxt\_e}}$ reference described above: We ignore the $e_{\text{nxt\_e}}$ reference during the SLR steps to the left of p, but we set it to e'' during the SLR steps to the right of p. Locating $p = p_{\max}$ in the example in Figure 2 illustrates the violation of the stronger invariant.

```
// e is a pointer to the initial edge connecting p_min and p_max.
// p is the point to be inserted next.
while (p.left_of(*(e->prv_p), *(e->nxt_p))) {
    if (e->inner) {
        if (p < *(e->prv_e->nxt_p))
            e = e->prv_e;
        else
            e = e->nxt_e;
    } else {
        if (p != *(e->prv_p) && p != *(e->nxt_p))
            // insert point p between e->prv_p and e->nxt_p
        break;
    }
}
// Either p has been removed, since it cannot appear on the
// LCH-polygon, or p has been inserted in the LCH-polygon.
```

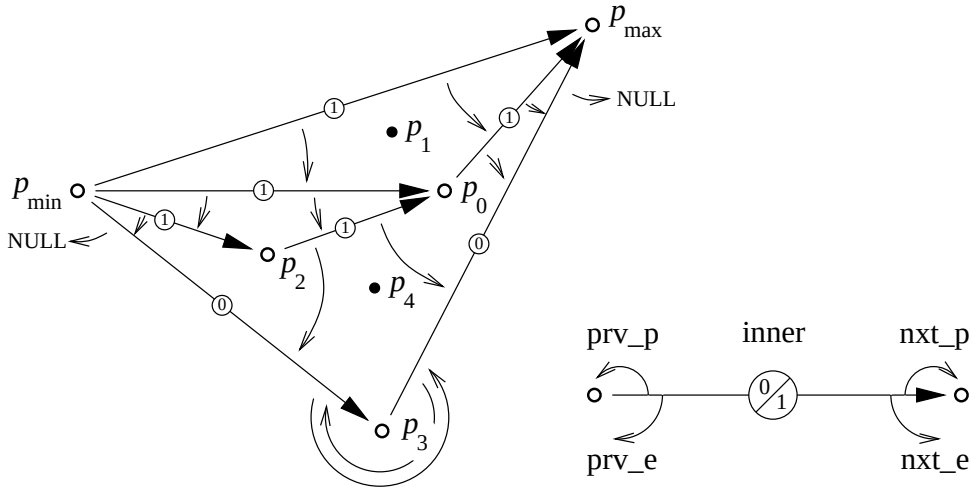

Figure 2: A possible configuration of the RIC data structure after the insertion of 5 points.

The following program fragment illustrates the SLR steps performed on the left side of p. On the right side, the additional assignment $e_{\text{nxt\_e}} = e''$ has to be considered. At the end of this fragment, all inner edges are updated and properly linked. We omit the description of the update of the two new edges, e' and e''.

```
// insert point p between e->prv_p and e->nxt_p.
// e1 and e2 are the two new edges on the LCH polygon.
l = e->prv_e;  // SLR on left side
while ( l != NULL && l->prv_p->left_of( p, *(l->nxt_p))) {
    l->inner = true;
    tmp = l;
    l = l->prv_e;
    tmp->prv_e = e1;
}
// similarly, perform SLR on the right side, incl. tmp->nxt_e = e2;
e->inner  = true;
e->prv_e  = e1;
e->nxt_e  = e2;
// the inner edges are updated.
```

If the input points occur in random order, the techniques of standard probabilistic analysis [CS89, Mul94, Sei93] give good expected time bounds, i.e., the average running time for all possible permutations of the input sequence. We define f_r as the expected number of points on the boundary of the convex hull of a random subsequence of length r. The expected running time, which is determined by the expected number of sidedness tests performed by the algorithm, is bounded by $O(n \sum_{r=2}^{n} f_r/r^2)$. This amounts to the worst case bound $O(n \log n)$, if $f_r = \Theta(r)$, but it becomes linear as soon as $f_r = O(r^{1-\varepsilon})$, $\varepsilon > 0$. For example, if the points are taken from a uniform distribution in a convex set, $f_r = O(\sqrt[3]{r})$. Or, if the points are taken from a uniform distribution in a convex polygonal area, $f_r = O(\log r)$ [RS63].

If we use a conservative implementation of the left-of predicate with the algorithm above, we still gain the following useful properties for the output:

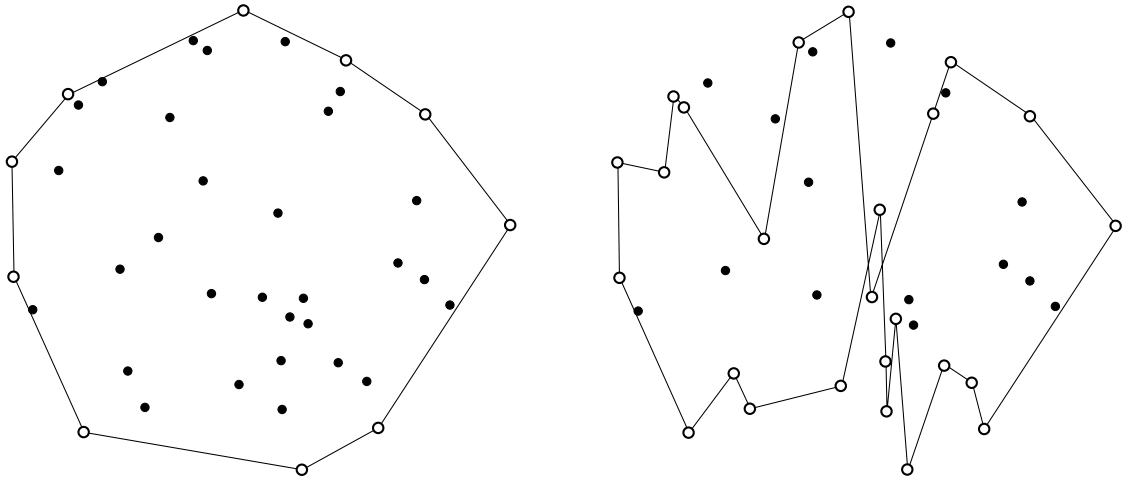

Figure 3: Two lower and upper convex hull polygons produced by the randomized incremental algorithm with conservative predicates. In both cases the used conservative implementation of the predicate with threshold was randomly distorted from true to false. To the left in 20%, to the right in 70% of the calls. Note that the result is still correct on the left side.

- *Observation* 1 *guarantees that we obtain an LCH-conserving polygon of the input points, since the invariant* $e_{prv\_p} \leq p \leq e_{nxt\_p}$ *needed for this observation remains true.*
- *The SLR-scheme can be applied to the output with an exact left-of predicate, which can check whether the sequence is already an LCH-polygon of* $\tilde{P}$*, or can perform further local repairs, otherwise.*

Note that we have lost the property of the SLR scheme, that all points of $\tilde{P}$ are on or above the output polygon. In fact, the polygon of the lower hull may even interfere with the polygon of the upper hull, see Figure 3.

3 Experimental results

We have made several experiments with our implementations to compare the efficiency of the different approaches. Here we varied:

- *The algorithm*: An implementation of successive local repair (SLR) following Graham's scan and randomized incremental construction with history graph (RIC).
- *The sidedness predicate used*: An ad-hoc implementation of the sidedness predicate as indicated in the introduction (AD-HOC), a conservative implementation of the predicate with static threshold (CONS), an adaptive floating point filter implementation by Shewchuk [She97] (SHE) and a static floating point filter by Fortune and Van Wyk [FW96] (FVW).
- *The point distribution:* random sets from a uniform distribution in a disk (•), on a circle (○), and random permutations of the complete $\sqrt{n} \times \sqrt{n}$-grid (⁝⁝).

SLR, μs per point after sorting

	predicate	10^2	10^4	10^6
●	ADHOC	0.86	0.82	0.72
	CONS	1.15	1.07	0.83
	SHE	1.64	1.00	1.05
	FVW	1.52	1.37	0.97
○	ADHOC	0.62	0.70	0.70
	CONS	0.81	0.85	0.77
	SHE	1.18	1.10	0.94
	FVW	1.05	0.98	2.28
⁝⁝	ADHOC	0.78	0.73	0.71
	CONS	0.93	0.86	0.80
	SHE	1.02	0.97	0.83
	FVW	4.67	4.22	3.72

RIC, μs per point after random shuffle

	predicate	10^2	10^4	10^6
●	ADHOC	1.31	1.15	1.16
	CONS	1.24	1.31	1.24
	SHE	1.81	1.71	1.45
	FVW	2.83	1.54	1.41
○	ADHOC	3.02	4.80	20.2
	CONS	2.79	4.59	19.2
	SHE	3.25	5.68	19.7
	FVW	3.73	4.94	23.6
⁝⁝	ADHOC	1.19	1.20	0.91
	CONS	1.30	1.24	0.98
	SHE	2.88	1.20	1.25
	FVW	5.52	1.83	1.20

Figure 4: Results of the run time experiments for input sets of 10^2, 10^4 and 10^6 points in μs per point.

Figure 4 shows a number of run times that do not include the time needed for sorting for SLR, and for generating a random permutation for RIC. The time needed for these preparatory tasks is by no means negligible. In fact, it usually dominates the algorithm: Using the quicksort algorithm in the Standard Template Library of C++, lex-sorting of 10^6 points took about 3.5 seconds. Again using STL, a random shuffle took about 2.5 seconds, which we could lower to one second with the random number generator made inline instead of a function call. Note that the sort is needed for the correctness of the SLR, but the random shuffle could be easily omitted if the input points are assumed to be sufficiently random. Anyway, these efforts are not influenced by the type of implementation of the sidedness predicate.

The numbers in the table are averaged over six experiments with different starting seeds for the random number generator; each experiment itself consisting of multiple runs. Averaged over these experiments, the average number of points that were on the convex hull was 15.8, 74.8 and 337.6 for 10^2, 10^4, and 10^6 points taken from the uniform distribution in a disk. The exact number of points on the convex hull for the points from the grid was 36, 396 and 3996.

The implementation. The test programs were written in C++ and compiled using Gnu `g++` version 2.8.1 with optimization level `-O3`. It was verified in the assembler output that the function inlining took place as intended, i.e., no function call is involved in the evaluation of the predicates. In fact, the whole SLR algorithm is a single function. Ignoring inlining would cost between 20% to 40% percent performance, if we would call a function to evaluate the predicate. However, for floating point filters, the exact evaluation should be separated as a function, while the filter remains inline. The timings were taken on a SUN UltraSPARC-II processor running on 248 MHz. As usual, timing results should be taken with care, even though the results were obtained using a sufficiently large sample of test runs and were reliable for small as well as for large numbers of input points, but could vary considerably for 1000 up to 10000 points, which we credit to interferences with the processor cache size. However, the observed differences were also consistent for these samples.

SLR

		10^2	10^4	10^6
●	*total*	380	39918	3999657
	exact	4	4	4
○	*total*	296	29996	2999975
	exact	4	28	962603
⁝⁝	*total*	360	39600	3996000
	exact	179	19799	1997999

RIC

		10^2	10^4	10^6
●	*total*	539	53145	5292522
	exact	40	68	134
○	*total*	926	182184	27356644
	exact	23	76	2578720
⁝⁝	*total*	556	44778	4442407
	exact	156	3131	48294

Figure 5: Results of the experiment counting predicate evaluations and exact predicate evaluations for input sets of 10^2, 10^4 and 10^6 points.

The SLR algorithm was used in a modified form with a sentinel, i.e., the first point q_0 duplicated, which protects the inner loop from underflow instead of the explicit test $k > 1$. Similarly the RIC algorithm was changed to work with two sentinels, both p_{min} and p_{max} were duplicated, such that the successive local repair steps are again protected against the boundary conditions. This makes the edge updates easier as well. The sentinels introduce artificial degeneracies – forcing the predicate to return false – which could influence timings for small input sets with exact predicates. To quantify this effect, we have counted the *total* number of evaluated predicates, and how often a filter based on our threshold would call the *exact* predicate, for one experiment with a fixed random number seed, see Figure 5. This table also explains the increase in running time observed for 10^6 points on the circle, since compared to 10^4 points many more exact evaluations were needed.

The predicates. We used for our conservative implementation a threshold of $t' = (3 \cdot 2^{-53} + 2^{-103})8B^2$, which is representable as an IEEE double precision number if B is a power of two. The FvW predicates were generated with the predicate compiler LN by Fortune and Van Wyk [FW96]. The static threshold was set to $t' = 2^{-48}B^2$, which is 4/3 times larger than the threshold used by us. The C++ output of the predicate compiler was slightly adapted to be comparable to our predicates, e.g. by inlining and by removing a superfluous conditional. The dynamic filter SHE by Shewchuk was available[11] in C and was adapted to C++ including inlining.

We also ran the algorithms with LEDA `rat_point`s and their exact and filtered predicates [MN95, Meh*97], which resulted in a slowdown by a factor between 4 for SLR and 40 for RIC compared to ADHOC. Of course, LEDA's implementation is a much more general tool than the other tailored implementations of predicates! In fact, in some settings, it was competitive with the FvW predicate (e.g. 10^6 points, SLR, ○).

Testing. Since we observe that the conservative predicate mostly behaves on our inputs just like their exact counterparts, testing with such input tells us little about the claims made in this paper. So we have artificially distorted the implementation of the predicates as follows: After the original conservative predicate decides to return *true*, we throw a biased coin that tells us to return *false* after all. The results in Figures 1 and 3 have been produced in this way. It is interesting to observe that SLR is much more sensitive to such a random distortion than RIC.

[11] `http://www.cs.cmu.edu/~quake/robust.html`

Conclusions. Conservative implementations of predicates can speed up algorithms if used instead of exact predicates. We see that compared to Shewchuk's predicate, which was the most competitive in the experiments, a conservative implementation gains about 20%, sometimes less. Compared to the static filter of Fortune and Van Wyk, a conservative implementation does not suffer from the degeneracies in the point sets chosen from the grid, which makes a difference of a factor of 4 for SLR. Interestingly, Shewchuk's predicate does not suffer either, which can be understood with a closer look at the implementation of the dynamic bounds: They can decide those degeneracies without resolving to the costly part of the predicate. Shewchuk itself used a tilted and – due to rounding – perturbed grid, which forced the predicates to perform the exact evaluation occasionally [She97]. However, we had not prepared such a test set.

4 Discussion

We have presented algorithms for convex hull and triangulation in the plane that are stable, even if they are used with a conservative implementation of the sidedness predicate. The output of the algorithms guarantees certain desirable properties with respect to the input. This is in contrast to so-called parsimonious algorithms, see [For89, Knu92]. These properties allow a simple checking and repair procedure with an exact predicate. Note that the checking does not have to be a complete verification process, since we can rely on certain properties, for example we never release vertices of the convex hull. The benefit of conservative predicates goes beyond a gain in run time efficiency. We experienced that they forced us to simplify programs by handling degeneracies implicitly instead of their explicit treatment.

Whether or not similar results can be obtained for problems, such as Delaunay triangulation or convex hull in higher dimensions, is not clear at this point and depends on the goal. Here, we were able to run a whole algorithm with the conservative left-of test. For more involved problems, it may still be beneficial to employ the right interleaved mixture of conservative and exact predicates. If so, the gain in efficiency will probably become more apparent in higher dimensions.

References

[Ame97] N. AMENTA, Computational geometry software. In [GO97] (1997) 951–960.

[Avn*97] F. AVNAIM, J.-D. BOISSONNAT, O. DEVILLERS, F.P. PREPARATA, AND M. YVINEC, Evaluating signs of determinants using single-precision arithmetic, *Algorithmica* **17** (1997) 111–132.

[Bak*97] J. E. BAKER, R. TAMASSIA, AND L. VISMARA, GeomLib: Algorithm engineering for a geometric computing library, (1997). (Preliminary report).

[Ber*97] M. DE BERG, M. VAN KREVELD, M. OVERMARS, AND O. SCHWARZKOPF, *Computational Geometry – Algorithms and Applications*, Springer Verlag (1997).

[BT86] J.-D. BOISSONNAT AND M. TEILLAUD, A hierarchical representation of objects: The Delaunay tree, *in* "Proc. 2nd Ann. ACM Sympos. Comput. Geom." (1986) 260–268.

[BY98] J.-D. BOISSONNAT AND M. YVINEC, *Algorithmic Geometry*, Cambridge University Press (1998).

[Bur*95] CH. BURNIKEL, J. KÖNNEMANN, K. MEHLHORN, ST. NÄHER, ST. SCHIRRA, AND CH. UHRIG, Exact Geometric Computation in LEDA, *in* "Proc. 11th Ann. ACM Sympos. Comput. Geom." (1995) C18–C19.

[CS89] K. L. CLARKSON AND P. W. SHOR, Applications of random sampling in computational geometry, II, *Discrete & Computational Geometry* **4** (1989) 387–421.

[DP98] O. DEVILLERS AND F. P. PREPARATA, A probabilistic analysis of the power of arithmetic filters, *Discrete & Computational Geometry* (1998), to appear.

[Ede87] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer Verlag (1987).

[Fab*98] A. FABRI, G.-J. GIEZEMAN, L. KETTNER, ST. SCHIRRA, AND S. SCHÖNHERR, On the Design of CGAL, the Computational Geometry Algorithms Library, *in* Algorithm Engineering (D. Wagner, Ed.), Trends in Software, John Wiley & Sons, (1998), to appear. Preliminary version available as technical report #291, Departement Informatik, ETH Zurich, Switzerland, February (1998).

[For89] S. FORTUNE, Stable maintenance of point set triangulations in two dimensions, *in* "Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci." (1989) 494–505.

[FW96] S. FORTUNE AND C. J. VAN WYK, Static Analysis Yields Efficient Exact Integer Arithmetic for Computational Geometry, *ACM Trans. Graph.* **15**(3) (1996) July, 223–248.

[GO97] J. E. GOODMAN AND J. O'ROURKE, EDS., *Handbook of Discrete and Computational Geometry*, CRC Press (1997)

[Gra72] R. L. GRAHAM, An efficient algorithm for determining the convex hull of a finite planar point set, *Inform. Process. Lett.* **1** (1972) 132–133.

[Hig96] N. J. HIGHAM, *Accuracy and Stability of Numerical Algorithms*, SIAM (1996).

[IEEE] IEEE *Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA (1985).

[Kle97] R. KLEIN, *Algorithmische Geometrie*, Addison-Wesley (1997).

[Knu92] D. E. KNUTH, *Axioms and Hulls*, *Lecture Notes in Computer Science* **606** (1992).

[MN95] K. MEHLHORN AND S. NÄHER. LEDA, a Platform for Combinatorial and Geometric Computing, *Communications of the ACM* **38** (1995) 96–102.

[Meh*97] K. MEHLHORN, S. NÄHER, AND C. UHRIG. *The LEDA User Manual, Version 3.5*, Max-Planck-Insitut für Informatik, 66123 Saarbrücken, Germany, (1997).

[Meh84] K. MEHLHORN, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer (1984).

[Mul94] K. MULMULEY, *Computational Geometry – An Introduction Through Randomized Algorithms*, Prentice Hall (1994).

[ORo94] J. O'ROURKE, *Computational Geometry in C*, Cambridge University Press (1994).

[PS85] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry – An Introduction*, Springer Verlag (1985).

[RS63] A. RÉNYI AND R. SULANKE, Über die konvexe Hülle von n zufällig gewählten Punkten, *Z. Wahrscheinlichkeitstheorie* **2** (1963) 75–84.

[SU98] J.-R. SACK AND J. URRUTIA, EDS., *Handbook in Computational Geometry*, to appear (1998).

[Sei93] R. SEIDEL, Backwards analysis of randomized geometric algorithms, *in* "New Trends in Discrete and Computational Geometry" (J. Pach, Ed.), Algorithms and Combinatorics 10, Springer Verlag (1993) 37–67.

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

[Ste74] P. H. STERBENZ, *Floating-Point Computation*, Englewood Cliffs (1974).