Experimental Comparison of the Cost of Approximate and Exact Convex Hull Computation in the Plane*

Stefan Schirra† Jan Tusch†

Abstract

We report on experimental studies regarding the cost of exact and approximate convex hull computation in the plane in order to find out how much time you can save by the transition to approximations. Our experiments let us arrive at the conclusion that in most cases adaptive exact computation based on double expansions as introduced to computational geometry by Shewchuk [13] is as fast as stable computation of approximate hulls as described by Jaromczyk and Wasilkowski [7].

1 Introduction

Convex hull computation in the plane is a computational geometry problem that has been frequently studied. Many design principles for geometric algorithms have been successfully applied to this problem, resulting in a number of optimal $O(n \log n)$ algorithms. Unfortunately, according to [7], most of these algorithms are numerically unstable if finite precision floating-point arithmetic is used as a straightforward substitute for exact real arithmetic, i.e., the real RAM. Examples for such failures of convex hull algorithms caused by floating-point arithmetic are presented and analyzed in [9]. However, convex hull computation in the plane is a rare case where numerical robustness issues have been intensively studied and where reliable robust algorithms dealing with the imprecision of floating-point arithmetic have been developed.

Li and Milenkovic [10] present an $O(n \log n)$ algorithm that computes a so-called ε-strongly convex δ-hull. The sequence of hull vertices forms a convex polygon even after perturbation of the vertices by at most ε. The hull does not necessarily contain all input points, but no point lies further than δ away from the computed hull. Guibas, Salesin, and Stolfi [6] use the framework of Epsilon Geometry [5] to present a worst-case $O(n^3 \log n)$ time algorithm that computes an ε-strongly convex δ-hull with better error bounds. Fortune [3] shows how to carefully implement Graham's scan [4] such that the resulting algorithm is stable: However, the computed hull is only approximately convex. Then, Jaromczyk and Wasilkowski [7] show how to "modify" Fortune's method such that the computed hull is a convex δ-hull. Actually, the modification is just a post-processing of the computed approximation where some points are kicked out.

On the other hand, the various convex hull algorithms designed with a real RAM in mind use only geometric predicates with low arithmetic demand. Therefore the exact geometric computation paradigm can be easily applied to these predicates using floating-point filters and exact arithmetic or other techniques [11, 13] ensuring exact decisions [15]. These exact decisions guarantee that the computed hull is the correct hull for the given input. Of course, such a hull is not necessarily strongly convex.

Compared to a naïve replacement of exact real arithmetic used in theory by floating-point arithmetic both approaches ensuring (increased) stability as well as approaches based on the exact geometric computation paradigm involve additional cost. In this paper we report on an experimental comparison of representatives of the two approaches. In our experiments, we use CGAL's generic convex hull code and CGAL's random points generators [2].

In Section 2 we present the competing methods. Section 3 describes the generator for the test data and section 4 presents the results of our experimental comparison. We conclude with section 5.

2 The Competitors

CGAL's generic convex hull code allows us to use different implementations of a geometric predicate with the same algorithm template. Fortune's careful implementation of Graham's scan is intrinsically Andrew's variant [1] with a particular implementation of the left turn predicate called *TriangleTest* in [3]. Therefore, we use CGAL's implementation of Andrew's variant of Graham's scan with different implementations of the left turn test. Besides the left turn, the algorithm uses comparison of Cartesian coordinates only in order to sort the input points lexicographically. Note that such comparisons are always exact. We use three different left turn implementations: (1) CGAL's default version for double precision floating-point coordinates, (2) a left turn test

*Partially supported by DFG grant SCHI 858/1-1

†Department of Simulation and Graphics, Faculty of Computer Science, Otto von Guericke University Magdeburg, Germany. {stschirr,tusch} at isg.cs.uni-magdeburg.de

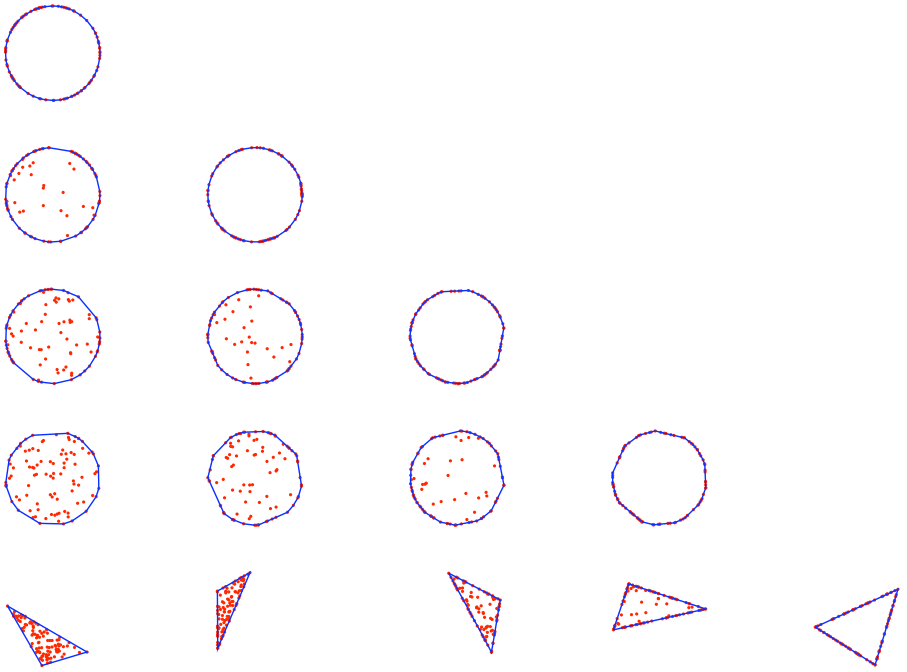

Figure 1: Sample point sets with 100 points each, and the corresponding hulls. n_d increases from left to right, n_e increases from bottom to top.

based on [13], and (3) a general purpose version of *TriangleTest* which does not require that the points passed to the test are already sorted. However, since points are sorted in Andrew's variant of Graham's scan, we provide a more efficient version (4) with another implementation of *TriangleTest* as well where we assume that the points are already passed to the predicate in a certain order. Here, we had to use a slightly modified version of the CGAL code for asserting the sorted order precondition. Finally, (5) we tested the combination of (4) with an implementation of the post-processing step described in [7]. Roughly speaking, the post-processing eliminates points where convexity is questionable with floating-point arithmetic.

Versions (3) and (4) are stable [3, 8], but compute a not necessarily convex approximate hull. In (5), the non-convexity defect is remedied by a post-processing resulting in a convex δ-hull. Version (2) delivers the exact hull for the given input. It uses Shewchuk's code [14] for 2d orientation. Shewchuk represents an arbitrary precision value x as a floating-point expansion, i.e., as a list of floating-point values $x_1, \ldots, x_k$, such that $x = x_1 + \cdots + x_k$. His implementation of the orientation predicate is adaptive, i.e., the cost is higher for more degenerate point configurations. In (2) as well as in (5) we assume that neither underflow nor overflow occurs. Version (1) is added as a reference only.

3 The Test Scenario

The design of the test point generation is guided by the following observations: The cost of the post-processing step à la [7] is significant only if the computed approximate hull has many vertices. The exact left turns are adaptive, i.e., the cost is higher for degenerate and nearly degenerate point triples. Since both the expected number of extreme points and the number of degeneracies are fairly small if the points are chosen at random from a uniform distribution in a disk or a k-gon for some constant k, we generate point sets with many extreme points and point sets with many nearly degenerate tests as well as intermediates. More precisely, in order to generate a set S of n distinct points, we generate a set S_e of n_e points on a circle, a set S_i of n_i points inside the convex hull of S_e, and a set S_d of n_d points (almost) on the segments of the convex hull of S_e, where $n = n_e + n_i + n_d$. We use CGAL's point generators `Random_points_on_circle_2`, `Random_points_on_segment_2`, and `Random_points_in_disk_2` to generate points on a circle, a segment, and in a disk, respectively. The circle is centered at the origin and has radius 10.0.

Fig. 1 shows sample point sets for $n = 100$ arranged in a matrix-like fashion. For the point sets in the bottom row we have $n_e = 3$. Along the vertical axis n_e increases to $0.25n$, $0.5n$, $0.75n$, and finally n, from bottom to top. In the first column $n_d = 0$. Along the horizontal axis

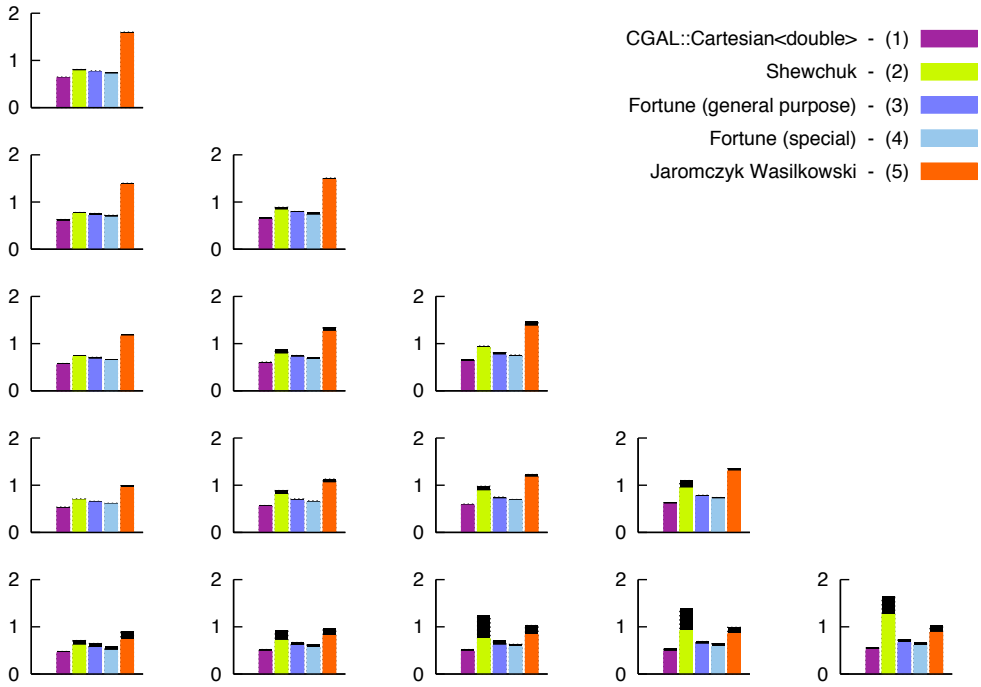

Figure 2: Running times for 100 points. The cumulative running time in seconds is shown for 10,000 iterations. In each test case, 10 different point sets were generated. The black parts on top of the bars represent the observed variability. Their lower border indicates the minimum running time measured and the upper border the maximum.

n_d increases from 0 to $0.25n$, ..., $0.75n$, and finally to $n-3$. In any case, $n_i = n - n_e - n_d$. Thus, on the diagonal from the upper left corner to the lower right corner, we have $n_i = 0$, and in the lower left corner we have $n_i = n - 3$.

4 Results

The C++ code was compiled with g++ 3.3.3 with optimization level `-O2` on a SUN running Solaris 2.9 and executed on that platform.

Fig. 2 visualizes running times for point sets with $n = 100$ like those shown in Fig. 1. Each bar summarizes running times for 10 different input sets. As expected the running times of version (2) using Shewchuk's orientation code grow along the horizontal axis, with increased variability due to the adaptiveness of the method. Furthermore, the running times of (5) grow with the number of extreme points. Since the general purpose version of the triangle test involves additional steps to ensure order preconditions of the actual triangle test, (4) is always faster than (3).

Fig. 3 visualizes running times for point sets with $n = 10,000$. These results and results for $n = 1000$ (not shown here) are analogous to those for $n = 100$.

And the winner is .... Well, that depends. If the number of extreme points is large and the point set is not nearly degenerate, the exact method is faster. At best, version (2) took only 49.4% of the running time of version (5). Having said that, if the point set is (nearly) degenerate and has few extreme points only, the approximate methods are faster. In our tests, version (2) took at most 178.4% of the running time of version (5). Besides the extreme cases, approximate and exact methods achieve fairly similar running times for convex hull computation in the plane. The savings with approximate computation are marginal at most. Note however, that we compared stable methods with the fastest exact methods we had at hand, cf. [12]. Other methods for exact geometric computation might result in (much) slower running times. For example, CGAL's implementation of Andrew's variant of Graham's scan with the left turn provided by CGAL's exact-predicates-inexact-constructions kernel is already slower than the tested version (2), especially if the interval filter stage fails. First experiments show that this holds for the exact sign of sum algorithm proposed for planar convex hull computation in [11] as well, even in combination with a floating-point filter. On the other hand, if the input coordinates are integers of known size, static filters could be used to speed up the exact computation even further.

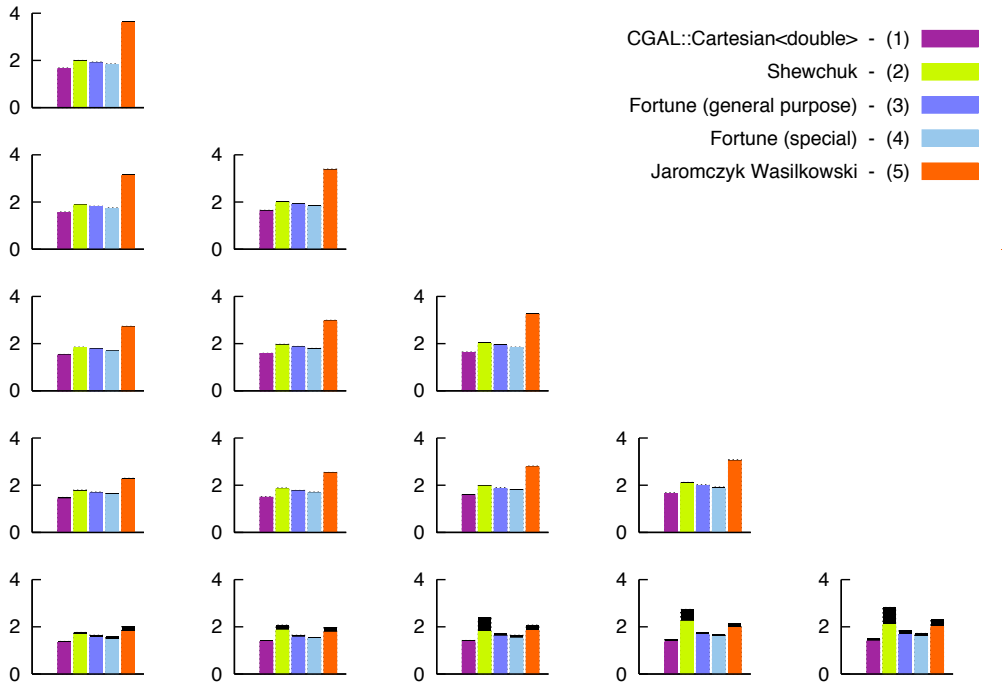

Figure 3: Cumulative running times for 10,000 points, 10 different point sets, 200 iterations.

5 Conclusions and Future Work

Our tests show that stable approximate convex hull computation is not faster than fast methods for exact convex hull computation in most cases. If most of the points are extreme, the exact methods are even superior. Regarding future work, it remains to add implementations of [6] and [10] to the test suite. At least, these algorithms produce strongly convex hulls which might be advantageous if the hull is used for further floating-point based computations where convexity is crucial.

References

[1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.

[2] CGAL (Comput. Geom. Alg. Library). www.cgal.org.

[3] S. Fortune. Stable maintenance of point set triangulations in two dimensions. *30th FOCS*, pp. 494–505, 1989.

[4] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[5] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. *5th SoCG*, pp. 208–217, 1989.

[6] L. Guibas, D. Salesin, and J. Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. *1st SIGAL*, LNCS 450, pp. 261–270, 1990.

[7] J. W. Jaromczyk and G. W. Wasilkowski. Computing convex hull in a floating point arithmetic. *Comput. Geom.: Theory and Appl.*, 4:283–292, 1994.

[8] D. Jiang and N. F. Stewart. Backward error analysis in computational geometry. *6th Workshop on Comput. Geom. and Appl.*, 2006.

[9] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. K. Yap. Classroom examples of robustness problems in geometric computations. *ESA04*, pp. 702–713, 2004.

[10] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. *Algorithmica*, 8:345–364, 1992.

[11] H. Ratschek and J. G. Rokne. Exact and optimal convex hulls in 2d. *Int. J. Comput. Geometry Appl.*, 10(2):109–129, 2000.

[12] S. Schirra. A case study on the cost of geometric computing. *ALENEX99*, LNCS 1619, pp. 156–176, 1999.

[13] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18(3):305–363, 1997.

[14] J. R. Shewchuk. Companion web page to [13]. www.cs.cmu.edu/~quake/robust.html.

[15] C. K. Yap. Robust geometric computation. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Comput. Geom.*, 2nd edition, 2004.