On the Design and Performance of Reliable Geometric Predicates using Error-free Transformations and Exact Sign of Sum Algorithms[*]

Marc Mörig[†] Stefan Schirra[†]

Abstract

We study the relevance of algorithms for exact computation of the sign of a sum of floating-point numbers and error-free transformations of arithmetic expressions on floating-point numbers for the design and implementation of low-dimensional geometric predicates. In a case study, we experimentally compare several implementations of planar orientation test and incircle tests that make use of such utilities.

1 Introduction

Floating-point computation is very fast, but also inherently inaccurate due to rounding errors. Naïvely applied, floating-point computation can cause fatal errors even with simple geometric algorithms [11], since floating-point based implementations of geometric predicates can produce wrong and inconsistent results. Most geometric predicates only involve sign computations for polynomial expressions in input coordinates or can at least be implemented like that. If all sign computations are correct, the predicate yields the correct result. The exact geometric computation paradigm now only asks for exactness in the sign computations, and not for computing exact values of the polynomial expressions. Configurations where a polynomial expression in a geometric predicate G evaluates to zero are called degenerate with respect to G.

Much progress has been made on the efficient implementation of reliable geometric predicates since the importance of the exact geometric computation paradigm has been widely acknowledged more than a decade ago. Efficient floating-point filters have been designed and combined to adaptive evaluation schemes for computing the sign of an arithmetic expression and are used in geometric software libraries like CGAL and LEDA. Floating-point filters compute a sign using fast floating-point arithmetic first and try to verify the computed sign, e.g. using an error bound computation. If the verification fails, they switch to some other method, for example, some exact arithmetic or a better filter, that uses higher precision in the floating-point computation or computes a better error bound. By cascading filters, sign computations can be made adaptive with respect to nearness to degeneracy. Prime examples of efficient cascaded floating-point filters are Shewchuk's predicates for orientation and incircle tests [20].

In the mid nineties Ratschek and Rokne presented an algorithm, called ESSA for exact sign of sum algorithm, to compute the sign of a sum of floating-point values exactly [15]. They advocated the use of their algorithm in computational geometry, see [6, 16, 17]. Using error-free transformations, any polynomial expression on floating-point numbers can be transformed into a sum of floating-point numbers. Thus an algorithm like ESSA is indeed a useful tool for the reliable implementation of a geometric predicate. Applications of summation algorithms in computational geometry are discussed by Graillat [7].

In Section 2, we look at some error-free transformations, especially those used in Shewchuk's predicates and ESSA. In Section 3 we briefly review ESSA and some more recent summation algorithms, that can be used to compute the exact sign of a sum, and discuss some modifications. Next, in Section 4, we illustrate the use of sign of sum algorithms and error-free transformations for the 2D incircle test. Finally, in Section 6, we report on the results of a case study, where we compare different reliable implementations of 2D incircle and orientation tests in CGAL's 2D Delaunay triangulation algorithm. Since initial floating-point filters for Delaunay predicates are already well studied [4, 5, 10, 20], we are mainly interested in nearly degenerate configurations. An input point set generator designed for this purpose is described in Section 5.

2 Error-Free Transformations

Error-free transformations transform an arithmetic expression involving floating-point numbers into a mathematically equivalent expression that is more suited for a particular purpose, e.g. sign computation. We use $\oplus, \ominus, \odot$ and $\oslash$ to denote the floating-point operations corresponding to $+, -, \cdot$ and $/$ respectively. We assume IEEE 754 double precision with exact rounding to nearest. Then, if neither overflow nor underflow occurs, the maximum relative error of a single operation is $\varepsilon = 2^{-53}$.

[*] Supported by DFG grant SCHI 858/1-1

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

The sum, $a + b$ can be transformed into $c^{\text{hi}} + c^{\text{lo}}$, such that $a \oplus b = c^{\text{hi}}$ and $a + b = c^{\text{hi}} + c^{\text{lo}}$. Note that c^{lo} is the rounding error involved in computing $a \oplus b$. Efficient algorithms for performing this transformation have been devised for IEEE 754 compliant arithmetic with exact rounding to nearest. The transformations are error-free unless overflow occurs. TWOSUM(a, b), due to Knuth [12], uses six floating-point additions and subtractions to perform this transformation, FASTTWOSUM(a, b), due to Dekker [2], requires $|a| \geq |b|$, but uses only three operations.

Analogously, TWOPRODUCT(a, b) computes floating-point values c^{hi} and c^{lo} with $a \odot b = c^{\text{hi}}$ and $a \cdot b = c^{\text{hi}} + c^{\text{lo}}$. TWOPRODUCT is due to Veltkamp and Dekker [2] and uses 17 floating-point operations. A substep consists of splitting a (and b) into $a = a^{\text{hi}} + a^{\text{lo}}$ where a^{hi} and a^{lo} use at most half of their mantissa. Thus the products $a^{\text{hi}} \odot b^{\text{hi}}$ etc. can be computed without rounding error. A single split takes 4 operations. Thus the cost for a sequence of TWOPRODUCT transformations can be reduced a bit if some numbers are involved in more than one product. TWOPRODUCT is error-free, unless overflow or underflow occurs.

The transformations above use standard IEEE 754 floating-point operations only and do not require explicit access to mantissa or exponent of a floating-point number. They are key ingredients in Shewchuk's efficient adaptive orientation and incircle predicate implementations [20].

There are also error-free transformations that involve bit-manipulation and/or exponent extraction. For example, Ratschek and Rokne [15] transform $a - b$ into $a' - b'$, with the requirement, that $\frac{1}{2}\varepsilon^{-1}b \geq a > 0$ and $\frac{1}{2}\varepsilon^{-1}b \geq a > 0$. The transformation has the property, that a' and $-b'$ have the same sign if neither of them is zero. If the exponents of a and b are equal, $a' = a \ominus b$ and $b' = 0$. Otherwise, if $a > b$, they set $a' = (a \ominus u)$ and $b' = (b \ominus u)$ where $u = 2^{\lceil \log_2 b \rceil}$. The remaining case $a < b$ for unequal exponents is handled analogously.

3 Summation Algorithms

Accurate summation of floating-point numbers has always been in the focus of research on numerical computations [8]. In this section we shortly address algorithms used in our predicate implementations to compute the sign of $s = \sum_{i=1}^{n} a_i$ where $a_1, \ldots, a_n$ are floating-point numbers.

ESSA iteratively performs error-free transformations on the largest positive and the smallest negative number in the current sum, thereby decreasing the sum of the absolute values of the summands. The iteration continues until the sum vanishes, or the largest positive number clearly dominates the sum of negative ones, or vice versa. In the original ESSA implementation [13], the error-free transformation described above is used. Both the set of positive summands and the set of negative summands are sorted initially. Apparently, it has not been observed so far, that creating a heap order is sufficient. In our implementation, we do not sort and use FASTTWOSUM for error-free transformation. Our modification, called "revised ESSA" later on, allows us to prove a better bound on the number of iterations, but there are also examples where the actual number of iterations grows. Both ESSA variants always return the correct sign, they are not affected by overflow or underflow. The number of summands is limited to $\frac{1}{2}\varepsilon^{-1} = 2^{52}$.

SIGNK is based on compensated summation, a well known approach to increase the accuracy of floating-point summation [9]. First we add the summands one by one using TWOSUM and store the results in the original variables. Besides an approximation a_n, this gives us a list of error terms $a_{n-1}, \ldots, a_1$. To improve the approximation we sum up a_1 to a_{n-1} on the fly and add a_n last, resulting in s'.

```
$s' = 0$
$\sigma = 0$
$c^{\text{hi}} = a_1$
for $i = 2, \ldots, n$ do
    $(c^{\text{hi}}, c^{\text{lo}}) = $ TWOSUM$(c^{\text{hi}}, a_i)$
    $a_{i-1} = c^{\text{lo}}$
    $s' = s' \oplus c^{\text{lo}}$
    $\sigma = \sigma \oplus |c^{\text{lo}}|$
$a_n = c^{\text{hi}}$
$s' = s' \oplus a_n$
```

Ogita et al. [14] analyze this algorithm and essentially prove the following error bound.

Theorem 1 ([14]) *Let s' and σ be computed by the algorithm above and*

$$\begin{aligned} \beta &= \sigma \odot \big(2\varepsilon n \oslash (1 - 2\varepsilon n)\big) \\ \Delta &= \varepsilon|s'| \oplus (\beta \oplus 2\varepsilon^2|s'|). \end{aligned}$$

Then, for $n < \frac{1}{2}\varepsilon^{-1} = 2^{52}$ and if neither overflow nor underflow occurs, we have $|s' - s| \leq \Delta$.

The original error bound [14] contains another term 3η, where η is the smallest positive floating-point number. This makes the bound valid also in the case underflow occurs in the computation of Δ. No underflow can occur in the computation of s'. We observed however, that the usage of the denormalized floating-point number 3η slows down the computation significantly.

The sign of s is positive if $s' > \Delta$ and negative if $s' < -\Delta$. If the error bound does not allow us to determine the sign, we eliminate the zeros among the a_i. If no summands are left, the sign is zero, otherwise we start over with the new a_i. The correctness of this iterated application of compensated summation combined with the error bound follows from Theorem 1. We next show that it also terminates.

Lemma 2 *If $n < \frac{1}{4}\varepsilon^{-1} = 2^{51}$ and neither overflow nor underflow occurs, then after a finite number of iterations $|s'| > \Delta$ and the algorithm terminates.*

Proof. Let $(c^{\mathrm{hi}}, c^{\mathrm{lo}}) = \textsc{twosum}(a, b)$. Then $c^{\mathrm{hi}} \oplus c^{\mathrm{lo}} = c^{\mathrm{hi}}$ and $|c^{\mathrm{lo}}| \leq \varepsilon|c^{\mathrm{hi}}|$, since c^{lo} is the rounding error involved in the computation $a \oplus b$. Furthermore also $|c^{\mathrm{hi}}| \oplus |c^{\mathrm{lo}}| = |c^{\mathrm{hi}}|$. If c^{hi} and c^{lo} have the same sign or $c^{\mathrm{lo}} = 0$ this holds since $|c^{\mathrm{hi}}| \oplus |c^{\mathrm{lo}}| = |c^{\mathrm{hi}} \oplus c^{\mathrm{lo}}|$. Now let us consider the case $c^{\mathrm{hi}} > 0$ and $c^{\mathrm{lo}} < 0$. Then $c^{\mathrm{hi}} + c^{\mathrm{lo}} < c^{\mathrm{hi}} < c^{\mathrm{hi}} - c^{\mathrm{lo}}$ and $c^{\mathrm{hi}} \ominus c^{\mathrm{lo}}$ will be rounded to c^{hi} since the distance from c^{hi} to the next larger floating-point number is at least as large as the distance to the next smaller floating-point number and ties are resolved using round-to-even. The remaining case follows by symmetry.

Each TWOSUM operation in the algorithm replaces a_{i-1} and a_i by the error and $a_{i-1} \oplus a_i$, in that order. Intuitively, the algorithm moves the more significant parts of the sum towards a_n. After some iterations we have $a_{i-1} \oplus a_i = a_i$ for $i = 2, \ldots, n$ and the TWOSUM operations will not lead to changes any more. Thus, at the end of that iteration, $s' = a_n$, $\sigma = |a_{n-1}|$ and $|a_{n-1}| \leq \varepsilon|a_n|$. It follows that

$$\begin{aligned}
\beta &= \sigma \odot \big(2\varepsilon n \oslash (1 - 2\varepsilon n)\big) \\
&\leq (1+\varepsilon)^2 |a_{n-1}| \frac{2\varepsilon n}{1 - 2\varepsilon n} \\
&\leq (1+\varepsilon)^2 \varepsilon |a_n| \frac{2\varepsilon n}{1 - 2\varepsilon n} \\
&\leq (1+\varepsilon)^2 \varepsilon |a_n|.
\end{aligned}$$

Here we use the fact that $a \circledcirc b \leq (1+\varepsilon)|a \circ b|$ for any operation $\circledcirc \in \{\oplus, \ominus, \odot, \oslash\}$. Furthermore

$$\begin{aligned}
\Delta &= \varepsilon|s'| \oplus (\beta \oplus 2\varepsilon^2|s'|) \\
&\leq (1+\varepsilon)\varepsilon|a_n| + (1+\varepsilon)^2\beta + (1+\varepsilon)^2 2\varepsilon^2|a_n| \\
&\leq (1+\varepsilon)\varepsilon|a_n| + (1+\varepsilon)^4\varepsilon|a_n| + (1+\varepsilon)^2 2\varepsilon^2|a_n| \\
&\leq 4(1+\varepsilon)^4\varepsilon|a_n| \\
&< |a_n| = |s'|.
\end{aligned}$$

□

SIGNK will also terminate if overflow or underflow occurs, although it may return a wrong sign. An underflow may only occur in the computation of the error bound Δ, making it smaller. This will not hinder termination. If an overflow occurs in TWOSUM, c^{lo} will be `NaN`. All subsequent computations involving c^{lo} will produce `NaN`s and hence Δ and s' will be `NaN` too. Comparisons involving `NaN`s always evaluate to false, so the algorithm will terminate if the error bound check is implemented as "NOT ($s' \leq \Delta$)" instead of "$s' > \Delta$". If an overflow occurs in the computation of s' or Δ, the algorithm may or may not terminate in the current iteration, but it will terminate in a later iteration, since it will end up in the case that TWOSUM operations do not lead to changes any more. Then no overflow occurs in the computation of s' and Δ.

ACCSIGN is based on the ACCSUM algorithm by Rump et al. [18]. ACCSUM computes one of the floating-point numbers adjacent to s. If s is itself a floating-point number or s is in the gradual underflow range, it is computed exactly. The algorithm uses error-free transformations, too, especially a splitting transformation. A summand a_i is split into ia_i^{hi} and a_i^{lo}, where $a_i = a_i^{\mathrm{hi}} + a_i^{\mathrm{lo}}$ and $|a_i^{\mathrm{lo}}| \leq M$ for some M that depends on the maximum magnitude and the number of summands. The split is done without accessing exponent or mantissa. The a_i^{hi} are summed up into an approximation s'. Thanks to the splitting and the choice of M this summation is free of rounding errors. If necessary, the remaining summands are summed up using the same technique and added to s'. For details we refer to [18]. Since some substeps in this iterative procedure are not necessary for computing the sign of s, they are omitted in ACCSIGN. ACCSIGN is not affected by underflow but the number of summands is limited to $\varepsilon^{-\frac{1}{2}-2} - 2 \approx 2^{26}$ which is not a restriction in practice.

4 Case Study Incircle Test

We illustrate the use of error-free transformations and corresponding sign of sum algorithms for the predicates of 2D Delaunay computation using CGAL's implementation based on the Delaunay hierarchy [1, 3]. In this section, we discuss the incircle predicate only, the orientation predicate is implemented analogously. The incircle predicate checks whether $s = (s_x, s_y)$ is contained in the circle through points $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$. This is tantamount to computing the sign of the determinant

$$\begin{vmatrix}
p_x & p_y & p_x^2 + p_y^2 & 1 \\
q_x & q_y & q_x^2 + q_y^2 & 1 \\
r_x & r_y & r_x^2 + r_y^2 & 1 \\
s_x & s_y & s_x^2 + s_y^2 & 1
\end{vmatrix}$$

Expanding the determinant results in a polynomial expression $p_x q_y r_x^2 + p_x q_y r_y^2 + \ldots$ consisting of 48 such subterms. By applying TWOPRODUCT six times, a single term is transformed into a sum of 8 floating-point numbers, resulting in a sum of 348 floating-point numbers. Let S_{348} be the corresponding arithmetic expression.

Alternatively, we consider

$$\begin{vmatrix} p_x - s_x & p_y - s_y & (p_x - s_x)^2 + (p_y - s_y)^2 \\ q_x - s_x & q_y - s_y & (q_x - s_x)^2 + (q_y - s_y)^2 \\ r_x - s_x & r_y - s_y & (r_x - s_x)^2 + (r_y - s_y)^2 \end{vmatrix}$$

corresponding to a translation that moves the point s to the origin. Using error-free transformations like $\textsc{twosum}(p_x, -s_x) = p_x^{\text{hi}} + p_x^{\text{lo}}$, we get

$$\begin{vmatrix} p_x^{\text{hi}} + p_x^{\text{lo}} & p_y^{\text{hi}} + p_y^{\text{lo}} & (p_x^{\text{hi}} + p_x^{\text{lo}})^2 + (p_y^{\text{hi}} + p_y^{\text{lo}})^2 \\ q_x^{\text{hi}} + q_x^{\text{lo}} & q_y^{\text{hi}} + q_y^{\text{lo}} & (q_x^{\text{hi}} + q_x^{\text{lo}})^2 + (q_y^{\text{hi}} + q_y^{\text{lo}})^2 \\ r_x^{\text{hi}} + r_x^{\text{lo}} & r_y^{\text{hi}} + r_y^{\text{lo}} & (r_x^{\text{hi}} + r_x^{\text{lo}})^2 + (r_y^{\text{hi}} + r_y^{\text{lo}})^2 \end{vmatrix}$$

Note that some of the $\Box^{\text{lo}}$ values might be zero, because the corresponding subtraction is exact. Since by Sterbenz lemma [21] a floating-point subtraction is exact if the operands differ by a factor of two in size at most, this is not unlikely! We transform

$$(p_x^{\text{hi}} + p_x^{\text{lo}})^2 + (p_y^{\text{hi}} + p_y^{\text{lo}})^2 =$$
$$p_x^{\text{hi}} p_x^{\text{hi}} + 2 p_x^{\text{hi}} p_x^{\text{lo}} + p_x^{\text{lo}} p_x^{\text{lo}} + p_y^{\text{hi}} p_y^{\text{hi}} + 2 p_y^{\text{hi}} p_y^{\text{lo}} + p_y^{\text{lo}} p_y^{\text{lo}}$$

and analogous terms into a sum using TWOPRODUCT, where a product incurring $\Box^{\text{lo}}$ is computed only if $\Box^{\text{lo}} \neq 0$. The resulting sum has between 4 and 12 summands. Then we use cofactor expansion on the last column. We transform

$$(q_x^{\text{hi}} + q_x^{\text{lo}})(r_y^{\text{hi}} + r_y^{\text{lo}}) - (q_y^{\text{hi}} + q_y^{\text{lo}})(r_x^{\text{hi}} + r_x^{\text{lo}})$$

and resembling expressions into a sum with between 4 and 16 summands. Each remaining product of two sums is then transformed into a single sum by applying TWOPRODUCT to each pair of summands, resulting in a sum with between 32 to 384 summands. Overall, we get an expression with between 96 to 1152 summands, called $S_{96:1152}$. We combine both expressions S_{348} and $S_{96:1152}$ with the sign of sum algorithms listed in the previous section, optionally with an initial float-filter from CGAL as discussed below.

We compare the resulting incircle predicate implementations with Shewchuk's adaptive incircle code [19, 20]. Shewchuk uses error-free transformations as well to transform the determinant into a sum. However, he uses a clever staged evaluation strategy that computes approximations with increasing accuracy, using so-called floating-point expansions. Comparison with an error bound is used to verify the sign of the current approximation. At each stage, some computations from previous stages are reused, leading to an adaptive predicate. Shewchuk also provides non-staged and hence non-adaptive predicate implementations. Furthermore, we compare with CGAL's *exact predicates inexact constructions kernel*. CGAL first uses a semi-static filter, and then a dynamic filter [4]. If both fail, CGAL's exact number type *MP_Float* is used. It is this well-engineered cascaded filter that we use in the initial phase of the filtered versions (ff ...) of our predicate implementations.

Remember that some of our predicate implementations based on error-free transformations are not guaranteed to work if overfow or underflow occurs. Implementations aiming for ease-of-use must therefore detect and handle overflow and underflow or use different techniques. Efficient detecting and handling such cases is a task for future work.

5 Test Data Generator

We are most interested in the performance of implementations of the incircle predicate for difficult instances, since state-of-the-art float-filters can be used to solve easy cases efficiently. In order to force a Delaunay triangulation algorithm to perform more difficult tests the generated test data contains points almost on a circle with no other points in its interior: First, we create a set $\mathcal{D}$ of d disks with a random radius $0 < r \leq r_{\max}$ and place a certain percentage f of the points (almost) on the boundary of their union, $\text{bd}(\cup\mathcal{D})$, cf. Table 1. Next, the remaining points are generated uniformly in the complement of the disks. All points are generated inside the unit circle. In order to get nearly degenerate point sets we use exact arithmetic to compute a point on a circular arc of $\text{bd}(\cup\mathcal{D})$ and then round it to a nearby floating-point point closest to the circular arc.

6 Results and Conclusions

We run experiments on both a PC with an Intel Core 2 Duo T5500 processor with 1.66 Ghz, using `g++ 4.1` and CGAL 3.2.1, and on a Sun Blade Station 1000 with 0.9 Ghz, using `g++ 3.3.3` and CGAL 3.2. Average running times for 25 input sets with 10 000, 1 000 and 100 points each are shown in Tables 1 - 3 respectively. Timings for the most competitive algorithms are visualized there as well. Interestingly, the rankings depend on the size of the point sets and also differ for the two platforms. The experiments show that an initial filter step is absolutely necessary, because otherwise the predicates based on the sign of sum algorithms, which are apparently non-adaptive, are too slow for non-degenerate point sets. For point sets without degeneracies, Shewchuk's adaptive predicates and all predicates using CGAL's cascaded filter perform equally well. This shows that for such input sets, the sign can almost always be deduced by the filter. For the other point sets, the rankings do not depend on the number of degenerate configurations. Original ESSA is obviously not competitive whereas our revised version comes close to the best ones, especially on the Sun platform. Predicates based on $S_{96:1152}$ perform relatively better on larger input sets. Because of the locality of the Delaunay triangulation algorithm and

Sterbenz' lemma, the fraction of sign evaluations of type $S_{96:1152}$ where the actual number of summands is small increases with the size of the input point sets, as illustrated in Figure 1.

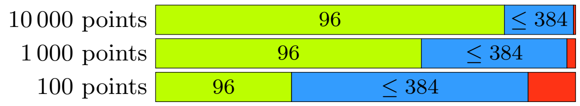


Figure 1: Fraction of sign evaluations of type $S_{96:1152}$ with 96, with at most 348, and with more than 348 summands for the generated input sets.

For point sets of 10 000 points, filtered SIGNK for $S_{96:1152}$ performs best on both platforms. On Sun, Shewchuk's adaptive predicates are next, while they are at rank 4 on Intel. With decreasing size of the input sets, Shewchuk's adaptive predicates perform relatively better. While filtered SIGNK for $S_{96:1152}$ stays the fastest on Intel, on Sun for sets of 100 points Shewchuk's adaptive predicates perform somewhat better. Since the computation times for small input sets are generally small, overall, filtered SIGNK for $S_{96:1152}$ is a good choice for the implementation of the incircle predicate.

References

[1] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[2] T. J. Dekker. A floating-point technique for extending the available precision. *Num. Math.*, 18(2):224–242, 1971.

[3] O. Devillers. The Delaunay hierarchy. *Int. J. of Found. of Comput. Sci.*, 13(2):163–180, 2002.

[4] O. Devillers and S. Pion. Efficient exact geometric predicates for Delaunay triangulations. In *ALENEX 2003*, pages 37–44.

[5] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations. *Int. J. of Comput. Geom. and Appl.*, 5:193–213, 1995.

[6] M. Gavrilova and J. G. Rokne. Reliable line segment intersection testing. *Computer-Aided Design*, 32(12):737–745, 2000.

[7] S. Graillat. Applications of fast and accurate summation in computational geometry. Research Report 03, Laboratoire LP2A, Université de Perpignan, 2005.

[8] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2. edition, 2002.

[9] W. Kahan. Further remarks on reducing truncation errors. *Comm. of the ACM*, 8(1):40, 1965.

[10] M. Karasick, D. Lieber, and L. R. Nackman. Efficient delaunay triangulation using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, 1991.

[11] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C.-K. Yap. Classroom examples of robustness problems in geometric computations. In *ESA 2004*, LNCS 2321, pages 702–713.

[12] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art Of Computer Programming*. Addison-Wesley, 3. edition, 1997.

[13] G. Mackenbrock, H. Ratschek, and J. G. Rokne. Experimental reliable code for 2d convex hull construction. `http://pages.cpsc.ucalgary.ca/~rokne/convex/`.

[14] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM J. on Sci. Comput.*, 26(6):1955–1988, 2005.

[15] H. Ratschek and J. G. Rokne. Exact computation of the sign of a finite sum. *Appl. Math. Computation*, 99(2-3):99–127, 1999.

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

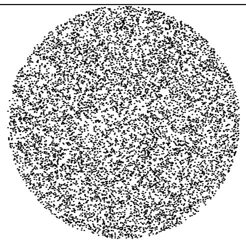
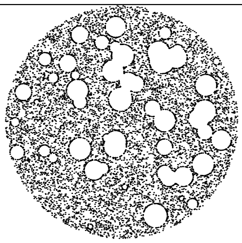
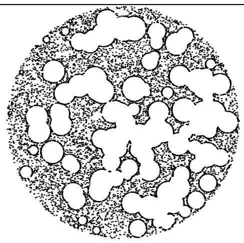
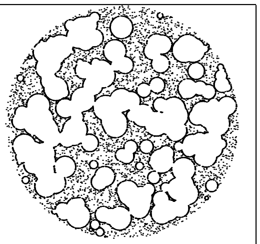
[17] H. Ratschek and J. G. Rokne. *Geometric computations with interval and new robust methods: applications in computer graphics, GIS and computational geometry*. Horwood Publishing, Ltd., 2003.

[18] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation. Technical Report 05.1, Faculty of Information and Communication Science, Hamburg University of Technology, 2005.

[19] J. R. Shewchuk. Companion web page to [20]. `http://www.cs.cmu.edu/~quake/robust.html`.

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

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

	 d=0, $r_{\max}$=0, f=0%		 d=63, $r_{\max}$=0.11, f=25%		 d=125, $r_{\max}$=0.11, f=50%		 d=188, $r_{\max}$=0.11, f=75%	
	Intel	Sun	Intel	Sun	Intel	Sun	Intel	Sun
org. ESSA S_{348}	7259	53005	7912	62692	8615	72819	9345	83577
rev. ESSA S_{348}	2028	5524	2199	5966	2392	6436	2588	6922
ACCSIGN S_{348}	598	2894	615	2968	636	3076	659	3195
SIGNK S_{348}	504	2749	505	2752	508	2763	510	2775
org. ESSA $S_{96:1152}$	1303	5274	1356	5875	1407	6431	1457	6999
rev. ESSA $S_{96:1152}$	526	1676	539	1726	552	1764	566	1813
ACCSIGN $S_{96:1152}$	260	1342	266	1366	272	1393	279	1433
SIGNK $S_{96:1152}$	208	1044	209	1047	210	1050	213	1057
Shewchuk non-adapt.	546	1943	546	1949	547	1966	552	1985
ff MP_Float (CGAL)	36	122	165	688	266	1116	344	1429
ff org. ESSA S_{348}	31	109	511	5665	885	9884	1177	12965
ff rev. ESSA S_{348}	32	110	162	455	264	719	343	914
ff ACCSIGN S_{348}	31	108	56	222	77	315	93	385
ff SIGNK S_{348}	31	108	49	203	63	279	75	336
ff org. ESSA $S_{96:1152}$	31	108	101	588	153	928	193	1177
ff rev. ESSA $S_{96:1152}$	32	108	55	180	74	237	88	279
ff ACCSIGN $S_{96:1152}$	31	110	43	159	51	200	59	233
ff SIGNK $S_{96:1152}$	31	109	39	148	46	182	53	211
ff Shew. non-adapt.	31	108	49	172	62	220	74	259
Shewchuk adaptive	34	106	52	153	67	198	81	234

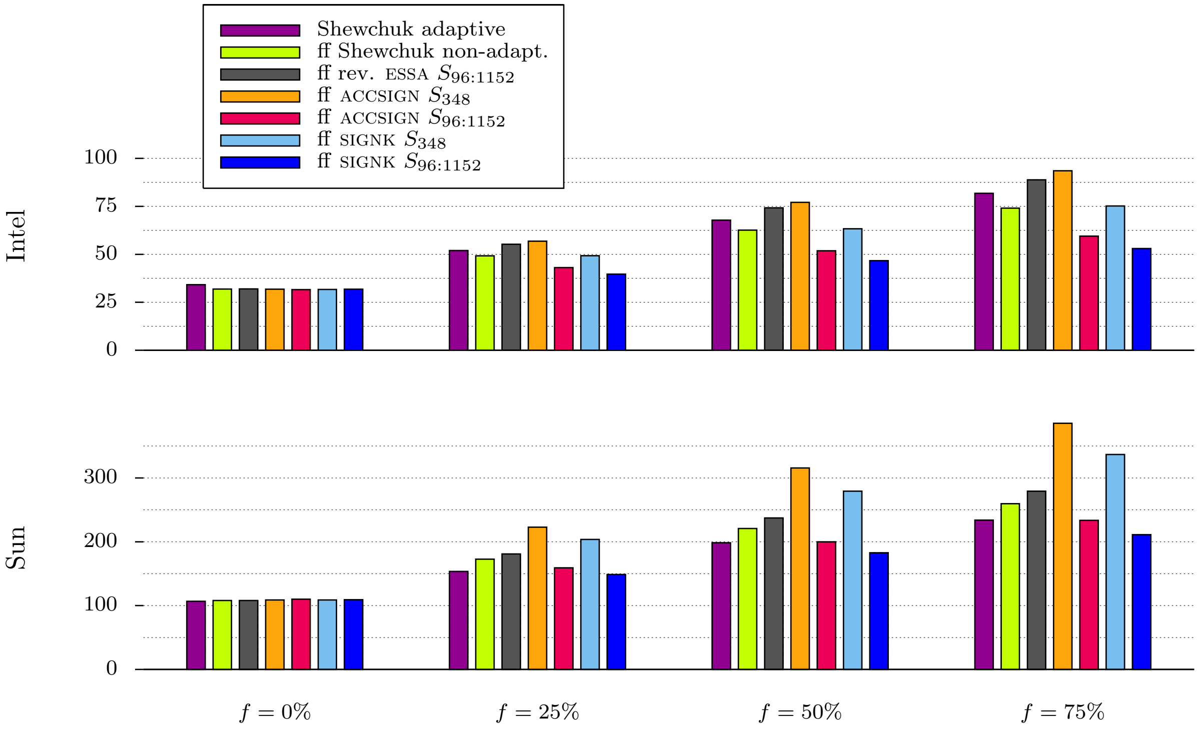
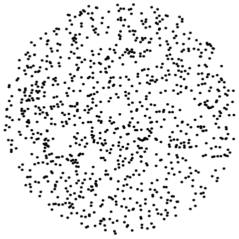
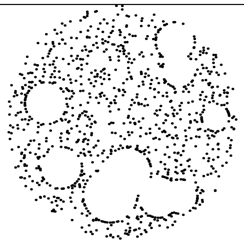
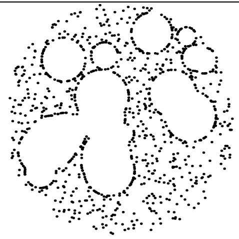
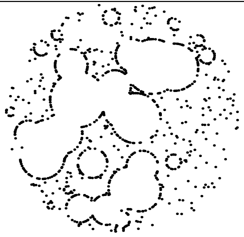


Table 1: Times in ms for computing the Delaunay triangulation of 10 000 points, averaged over 25 input sets.

	 $d=0$, $r_{\max}=0$, $f=0\%$		 $d=13$, $r_{\max}=0.24$, $f=25\%$		 $d=26$, $r_{\max}=0.24$, $f=50\%$		 $d=38$, $r_{\max}=0.24$, $f=75\%$	
	Intel	Sun	Intel	Sun	Intel	Sun	Intel	Sun
org. ESSA S_{348}	653	4616	701	5357	753	6198	817	6981
rev. ESSA S_{348}	182	497	194	529	208	568	225	602
ACCSIGN S_{348}	54	262	55	266	56	270	57	276
SIGNK S_{348}	46	253	46	253	46	254	46	252
org. ESSA $S_{96:1152}$	136	537	143	602	149	664	151	723
rev. ESSA $S_{96:1152}$	54	170	56	172	57	177	58	182
ACCSIGN $S_{96:1152}$	27	131	27	134	28	135	28	139
SIGNK $S_{96:1152}$	22	111	22	110	22	110	22	111
Shewchuk non-adapt.	52	175	52	175	52	177	51	177
ff MP_Float (CGAL)	2	9	19	78	34	143	45	193
ff org. ESSA S_{348}	2	9	62	650	114	1279	158	1760
ff rev. ESSA S_{348}	2	9	18	49	32	87	44	117
ff ACCSIGN S_{348}	2	9	5	22	8	34	10	44
ff SIGNK S_{348}	2	9	4	20	6	30	8	39
ff org. ESSA $S_{96:1152}$	2	9	13	80	21	140	27	185
ff rev. ESSA $S_{96:1152}$	2	8	6	18	8	27	11	33
ff ACCSIGN $S_{96:1152}$	2	9	4	15	5	21	6	26
ff SIGNK $S_{96:1152}$	2	9	3	14	4	18	5	22
ff Shew. non-adapt.	2	9	4	16	6	23	8	28
Shewchuk adaptive	2	8	4	13	6	18	7	22

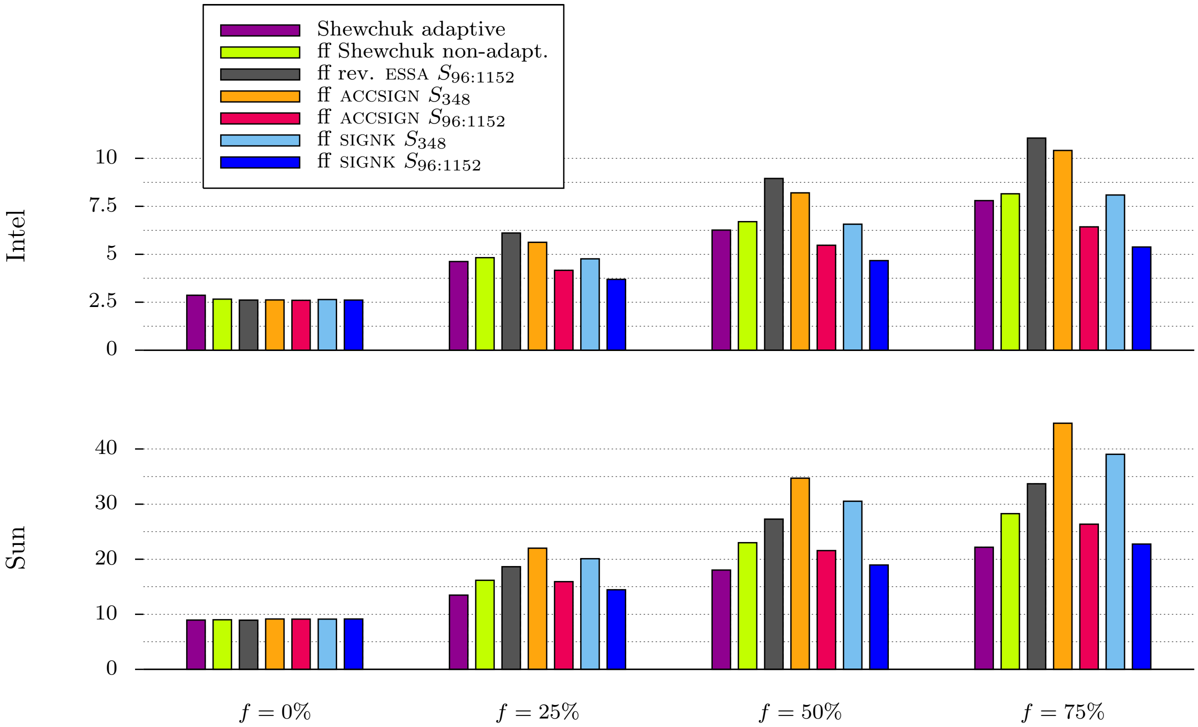
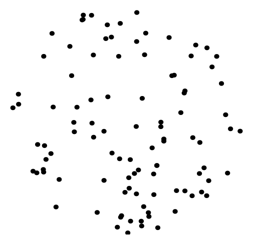
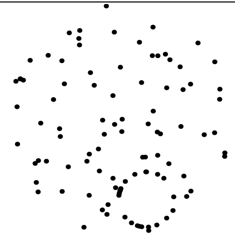
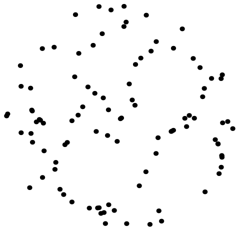
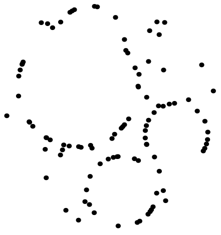


Table 2: Times in ms for computing the Delaunay triangulation of 1 000 points, averaged over 25 input sets.

	 d=0, $r_{\max}$=0, f=0%		 d=3, $r_{\max}$=0.5, f=25%		 d=5, $r_{\max}$=0.55, f=50%		 d=8, $r_{\max}$=0.53, f=75%	
	Intel	Sun	Intel	Sun	Intel	Sun	Intel	Sun
org. ESSA S_{348}	47.1	306.6	50.2	356.1	53.3	417.8	56.4	475.5
rev. ESSA S_{348}	13.0	35.4	13.8	37.8	14.6	40.1	15.4	42.9
ACCSIGN S_{348}	4.1	19.3	4.1	19.7	4.2	19.7	4.2	20.2
SIGNK S_{348}	3.5	19.2	3.5	19.3	3.5	19.0	3.4	19.1
org. ESSA $S_{96:1152}$	13.9	50.1	13.9	57.9	14.4	64.0	14.9	69.5
rev. ESSA $S_{96:1152}$	5.5	15.8	5.4	16.6	5.5	16.2	5.6	16.6
ACCSIGN $S_{96:1152}$	2.8	12.3	2.7	12.7	2.8	12.3	2.8	12.5
SIGNK $S_{96:1152}$	2.4	11.6	2.3	11.9	2.3	11.1	2.3	11.1
Shewchuk non-adapt.	4.2	13.1	4.2	13.3	4.1	12.9	4.1	13.1
ff MP_Float (CGAL)	0.2	0.7	1.6	6.5	3.2	14.7	4.5	19.9
ff org. ESSA S_{348}	0.2	0.7	4.8	52.0	10.2	122.4	14.6	170.5
ff rev. ESSA S_{348}	0.2	0.6	1.4	3.9	2.8	8.2	4.0	11.2
ff ACCSIGN S_{348}	0.2	0.6	0.4	1.8	0.7	3.2	0.9	4.2
ff SIGNK S_{348}	0.2	0.7	0.3	1.6	0.5	2.9	0.7	3.7
ff org. ESSA $S_{96:1152}$	0.2	0.7	1.3	7.5	2.4	19.3	3.5	24.9
ff rev. ESSA $S_{96:1152}$	0.2	0.6	0.6	1.6	0.9	3.0	1.3	3.8
ff ACCSIGN $S_{96:1152}$	0.2	0.7	0.4	1.3	0.5	2.3	0.7	2.8
ff SIGNK $S_{96:1152}$	0.2	0.7	0.3	1.2	0.4	2.0	0.5	2.4
ff Shew. non-adapt.	0.2	0.7	0.4	1.3	0.6	2.1	0.8	2.6
Shewchuk adaptive	0.2	0.7	0.3	1.0	0.5	1.4	0.6	1.8

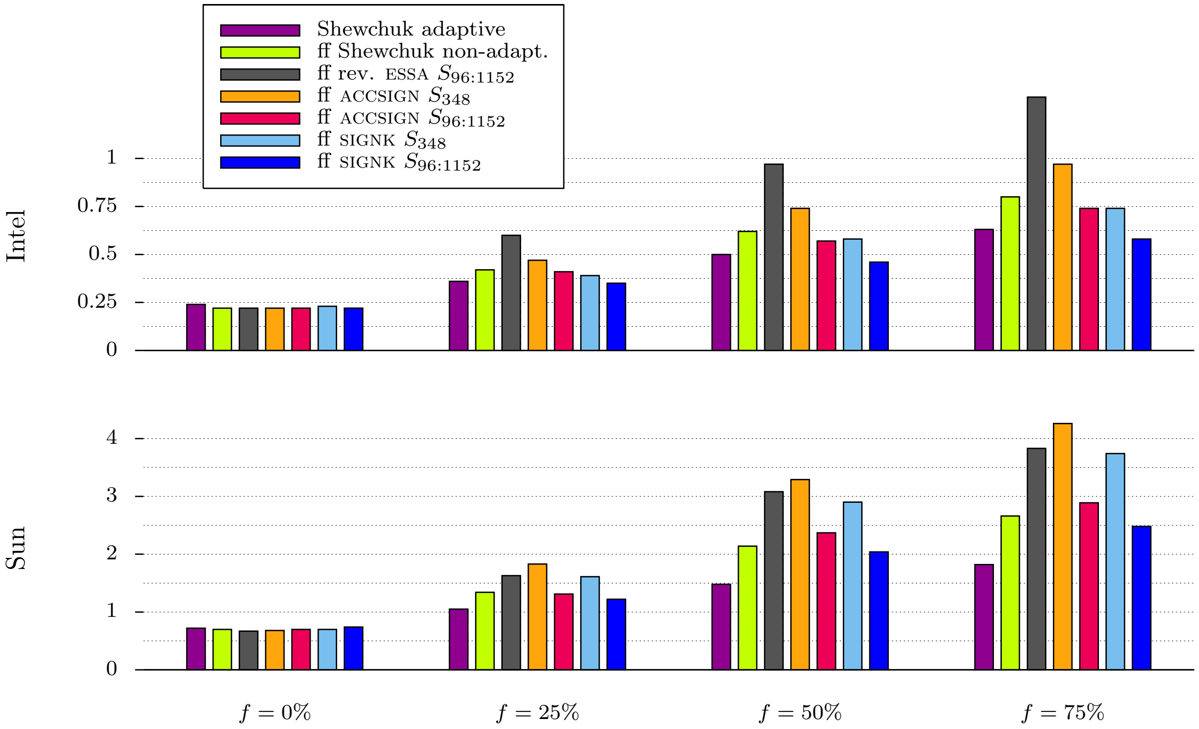


Table 3: Times in ms for computing the Delaunay triangulation of 100 points, averaged over 25 input sets.