Experimental Evaluation of Structural Filtering as a Tool for Exact and Efficient Geometric Computing *

Stefan Näher Martin Taphorn †

Abstract

We report on experimental studies and further investigations of *structural filtering* introduced in [3] as a paradigm for efficient exact geometric computing. In particular, we show that structural filtering can almost completely avoid the overhead of floating-point filters for many fundamental geometric problems. Furthermore we develop new repair strategies that avoid the very bad performance of the simple repair methods presented in the original paper in presence of difficult input data.

1 Introduction

Algorithms in computational geometry use geometric predicates in their conditionals. A common strategy for exact geometric computing is to evaluate all predicates exactly and to use *floating-point filters* ([2]) to make this evaluation efficient. This approach is used in the exact geometry kernels of CGAL and LEDA ([4]).

Floating-point filtering (also called predicate filtering) works as follows. The evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. The filter evaluates the expression using floating-point arithmetic and also computes an error bound to determine whether the floating-point result is reliable. If the error bound does not suffice to prove correctness, the expression is re-evaluated using exact arithmetic.

This strategy incurs an overhead when compared to a pure floating-point implementation. For *easy inputs* where the floating-point computation always yields the correct sign, the overhead consists of the computation of the error bound which is about a factor of two for good filter implementations. For *difficult inputs* where the floating-point filter fails very often, the overhead may be much larger, but this is not really relevant, as the floating-point computation will produce an incorrect result.

Structural filtering as introduced in [3] views the execution of an algorithm as a sequence of more general steps and applies filtering at the level of these steps. A step may contain many predicates and errors are allowed in the evaluations of these predicates, but finally, the outcome of each step has to be correct.

In order to achieve this correctness every step is executed using pure floating-point arithmetic in the first place. Then the result is checked for correctness and if this check fails, a repair procedure is called. The most simple repair strategy is to recompute the entire step using predicate filtering. We will see that often much more efficient repair strategies exist (see section 3).

In this context floating-point filtering is just a specialization of structural filtering where each step consists of one geometric predicate, checking for correctness is done by comparing with the error bound, and the repair step consists of re-computing the corresponding expression with exact arithmetic.

Structural filtering only works if every step is guaranteed to terminate even in presence of arbitrary errors in the floating-point evaluation of its predicates. In many cases this condition is obviously fulfilled because the underlying structure is acyclic, e.g. when searching in a binary search tree or skiplist. In other cases termination can be easily guaranteed by slightly modifying the algorithm, e.g. by marking all visited nodes when walking in a triangulation. The goal of structural filtering is to implement exact geometric computing at the cost of floating-point arithmetic in cases where no repairing step is necessary.

In this work we investigate experimentally the potential of structural filtering for different fundamental geometric problems and show that this goal can be achieved in many cases. In particular for new implementations of algorithms for sorting, convex hull, plane sweep, point location, and range searching.

The remainder of this paper is structured as follows. In Section 2 we give the details of the structural filtering methods we used for a collection of fundamental problems, such as sorting, search trees and skiplists, plane sweep, range trees, and point-location. Section 3 presents the point generators we used for easy and difficult problem instances and gives the results of the most important experiments. Finally, Section 4 gives some conclusions and reports on current and future work.

*This work was supported by DFG-Grant Na 303/2-1

†Department of Computer Science, University of Trier, Germany. {naeher, taphorn}@uni-trier.de

2 Structural Filtering and Repairing

2.1 Sorting

Sorting points according to different linear orderings (e.g. the lexicographic ordering of the cartesian or polar coordinates) is a basic step in many algorithms for geometric problems, as in the computation of convex hulls, triangulations, and many plane sweep algorithms. Quicksort has to be proven the most efficient sorting algorithms in these applications.

In [3] a structural filtering version of quicksort is presented that uses an exact insertion sort routine in the repair step. For the partitioning a cheap floating-point compare function is used. After the recursive calls of quicksort we end up with two increasingly sorted subsequences separated by the pivot element. However, due to possible errors made in the partitioning the entire sequence may not be sorted correctly. This can easily be tested by comparing the pivot with its two neighbors using the exact compare function. If the test fails the repairing is done by a call of insertion sort again using the exact compare function. Please see [3] for a more detailed description and an analysis of this algorithm.

In the experimental analysis in section 3 we compare this original repair strategy (which we call *simple repair*) with an improved strategy (*smart repair*). The new strategy defines an upper limit for the number of swaps executed by all insertion-sort calls. If this limit is reached the algorithms stops and starts from scratch calling quicksort with the exact compare function. The smart repair strategy avoids the very bad performance of insertion sort in presence of difficult input data (see section 3).

2.2 Sorted Sequences

Sorted Sequences are used in plane sweep algorithms to dynamically maintain the objects intersected by the sweep line at its current position (see [5] for details). They can be implemented by balanced binary search trees or skiplists ([7]). The most fundamental operation is the *locate* function that finds the position of a given objects among all existing objects.

In the structural filtering version of a search tree (or skiplist) $\log n$ cheap floating-point comparisons are used to find the (approximative) position of x among the leaves of the tree (see Figure 1). Since search trees are acyclic termination of the locate step is guaranteed even in presence of arbitrary bad errors in the compare function.

Checking whether the computed position p is correct or not can be done by at most 2 exact comparisons with the neighbor leaves. For the repair step we assume that the distance between p and the correct position is d.

In the *simple repair strategy* we use linear search starting at p to find the correct position. This takes d exact compares. For the *smart repair strategy* we use finger search which uses at most $\log d$ exact compares.

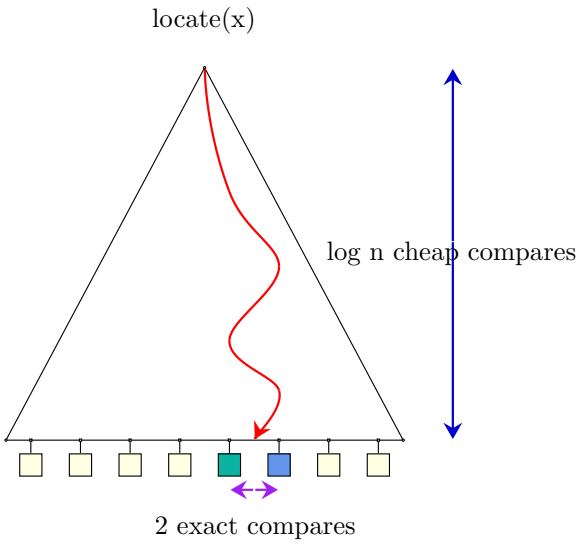

Figure 1: Search tree using structural filtering.

2.3 Multi-dimensional Search Trees

Multi-dimensional trees were first introduced in [1]. They exist in different variants, such as range-, segment-, interval- and priority-search trees. We concentrate on two-dimensional range-trees in this paper. Similar results can be obtained for the other variants.

Range trees consist of a primary binary tree data structure that stores secondary search trees in each node. Orthogonal range queries can be realized by locating the two search paths to the x coordinates in the primary tree and then perform a sequence of at most $2 \log n$ one-dimensional range queries on secondary search trees. As for simple search trees and skiplists termination is guaranteed since the underlying data structure is acyclic.

The *simple repair* strategy uses (as in the case of skiplists) a linear search to find the correct positions in the primary and secondary search trees and the *smart repair* stops linear searching in the secondary structures as soon as some upper bound for the total steps executed in all linear searches is reached. Then the query is repeated from scratch with predicate-filtering.

2.4 Point Location

We consider the problem of locating a point p in a given triangulation T. The well-known walking-algorithm presented in [8] and [5] starts in an arbitrary triangle of T and traverses the faces of T along a straight ray directed towards p until the triangle containing p is reached. In every step it uses the orientation predicate

to find an edge through which the ray leaves the current triangle. In the structural filtering version of this algorithm the cheap floating-point version of the orientation predicate is used during the walk. Since the underlying structure is not acyclic we have to guard against possible loops to guarantee termination of the walk. This can easily be done by marking every visited triangle. As soon as a marked triangle is visited again, the algorithm starts a new walk from the current position now using the exact orientation predicate. We use a simple repair strategy. The check of correctness is done by using the exact orientation predicate to find out whether the last triangle really contains point p. If it fails a new exact walk is started at the current position. This strategy behaves good for both simple and difficult input data (see section 3).

2.5 Delaunay Triangulations

We use the flipping algorithm presented in [9] to construct the delaunay triangulation of a given set of points S. This algorithms has shown to be very effective in practice. It starts with constructing an arbitrary triangulation of S and then applies a local transformation called flipping to the edges of the triangulation until every edge e fulfills the Delaunay property, i.e. the two triangles ajdacent to e have both empty circumscribing circles. The checking of the delaunay is done by a in-circle test applied to the corresponding vertices.

In the structural filtering version of this algorithm floating-point arithmetic is used in the in-circle tests. Note however that an edge e must not be flipped if the qudrilateral with diagonal e is not convex. Otherwise, the resulting structure would not be a legal triangulation. Since the in-circle test may be incorrect we have to test for convexity using exact arithmetic before the flipping can be done.

The repair step is simply done by calling the exact variant of the flipping algorithm (using exact arithmetic in the in-circle tests) on the possibly incorrect result of the first step. If the result was correct the repair step does nothing then checking all edges for the Delaunay property. If the result was wrong, errors are corrected by the exact flipping algorithm. Here the checking takes linear time and we cannot expect to be able to reduce the running time very close to the pure floating-point version of the algorithm. This is reflected in the experimental results of section 3.5.

2.6 Convex Hulls

Computing the convex hull of a point set S is a fundamental geometric problem. One of the most popular methods to solve this problem is the incremental or sweep algorithm (see [10]). It works as follows. Sort S from left to right and initialize the convex hull with the first three points. Now, add the remaining points one by one while maintaining the convex hull CH of the points visited so far by computing the two tangents from the current point p to CH. The cost of this algorithm is dominated by the running time of the initial sorting step. We can significantly reduce this cost by using the structural filtering variant of quicksort presented in section 2.1. See section 3.4 for experimental results.

2.7 Line Segment Intersection

The Bently-Ottmann sweep line algorithm ([11]) is a well-known and efficient method for computing the intersections of a set of straight line segment in the plane. In the implementation of this algorithm the sorted sequence data structure (see section 2.2) can be used to represent both the event queue und the set of segments intersecting the sweep line at its current position. See [5] for a detailed description and analysis. The running time of sweep algorithms in general is dominated by the cost of the operations applied to these sorted sequences. We were able to improve the running time of the segment intersection algorithm considerably by using the structural filtering variant of the sorted seqence data structure presented in section 2.2. This is reflected by the experimental results in section 3.4.

3 Experiments

In the experiments presented in this section we test our new structural filtering variants of the presented algorithms and data structures with two different kinds of input data.

easy input data: we randomly choose points with 30-bit integer coordinates in the square $[0 \dots 2^{30} - 1] \times [0 \dots 2^{30} - 1]$.

difficult input data: we generate difficult inputs in a systematic way by generating a point set as in the easy case and translating each point by adding 2^k to both x and y coordinates. Here k is a parameter increasing from 31 to 100. The effect of the translation is that the bits of the original coordinates are shifted to the right and as soon as k reaches 53 (the length of the mantissa in the IEEE 754 double floating-point format) we start to loose precision. This loss of precision introduces errors in the input data that grow with increasing values for k. When k reaches 83 no single bit of the original input data is available anymore and all coordinates will become equal.

With the easy input data we can show that our variants of the presented algorithms and data structures almost run as fast as pure floating-point implementations. We use the difficult input to show the robustness of our implementations in cases where many errors in the floating-point arithmetic happen. For these cases

the experiments show that the running time is basically the same as for floating-point filter implementations.

3.1 Quicksort

Figure 2 shows the result for easy input data. We see (as expected) that quicksort with structural filtering as almost as fast the pure floating-point version of the algorithm, which is about as two times as fast as the floating-point filter version.

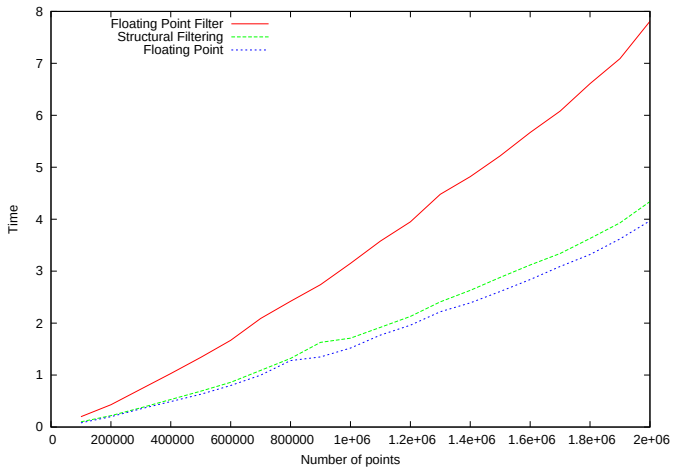

Figure 2: Quicksort with easy input.

Figure 3 shows the results for difficult input data. The diagram shows the performance of both simple and smart repair strategies and compares them to the floating-point filter version. We can see that running times increase as soon as the error parameter k reaches the value of 53 bits and that the simple repair strategy completely degenerates for very large values of k. The smart repair strategy however behaves much better for larger k and reaches about the same performance as the floating-point filter version in this case.

3.2 Sorted Sequences

For the experiments on sorted sequences we use the skiplist implementation from the LEDA library (see[5]).

Figure 4 shows the results for easy input data. We perform 500,000 locate operations on a sorted sequence containing a increasing number of points. As in the case of quicksort we again can considerably reduce the overhead of the floating point filter version and almost reach the performance of the pure floating point version.

Figure 5 shows the results for difficult input. For increasing values of the error parameter k the smart repair strategy (using finger search in the repair step) performs much better than the simple repair strategy (using linear search).

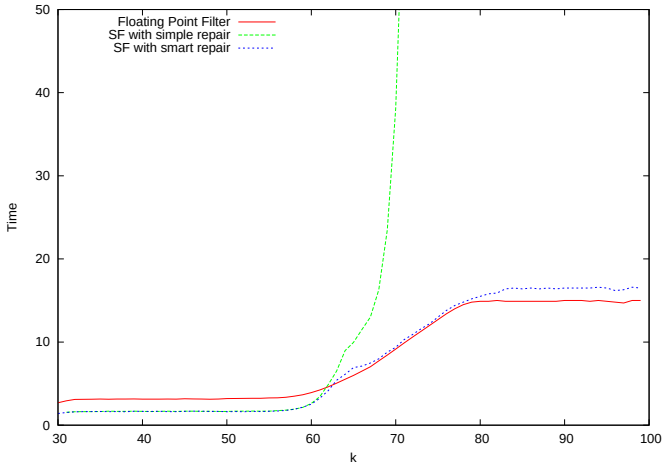

Figure 3: Quicksort with difficult input.

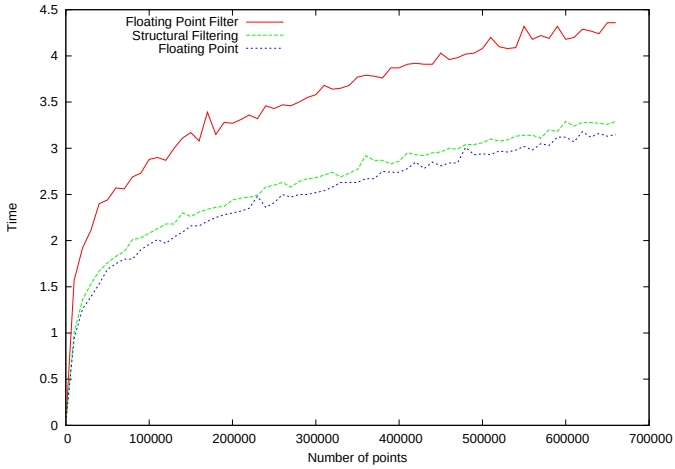

Figure 4: Sorted sequence with easy input.

3.3 Range Trees

Figure 6 shows the results for easy input data. As in the examples before, we can reach almost the same running time as the pure floating-point version of the tree.

Figure 7 shows the results for difficult input. Here again we can see the very bad behavior of the simple repair strategy in presence of a large number of errors and the much better performance of the smart repair procedure.

3.4 Point Location, Segment Intersection, and Convex Hull

We present the experimental results for our new variants of the point location, segment intersection, and convex hull algorithms, as described in section 2. Figures 8, 10 and 12 show the results for easy input data. In all cases we can reduce the overhead of the floating point filter version and can almost reach the performance of

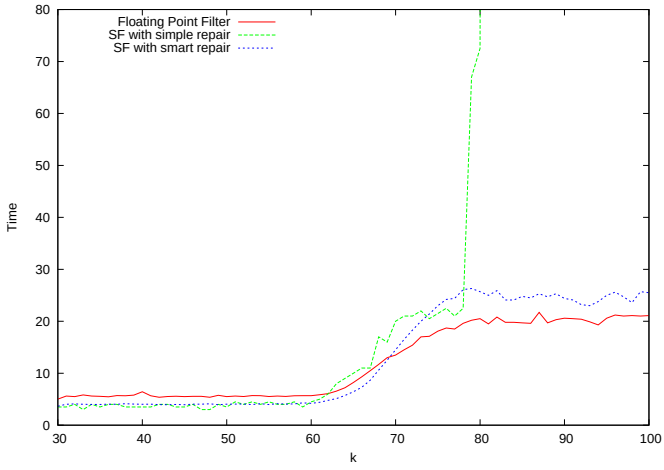

Figure 5: Sorted sequence with difficult input.

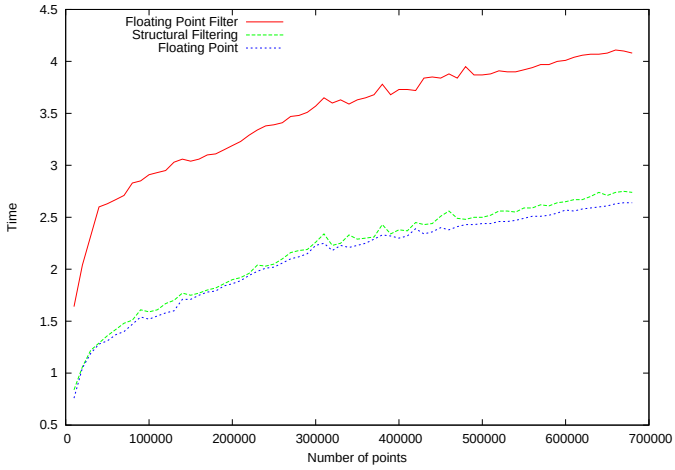

Figure 6: Range Tree with easy input.

the pure floating-point version.

Figures 9, 11, and 13 shows the results for difficult input data. As one can see, our repair strategies behaves good for difficult input data and reaches about the same performance as the floating-point filter version.

3.5 Delaunay Triangulation

In section 2.5 we presented a new structural filtering variant of the flipping-algorithm to compute the Delaunay triangulation of a set o points in the plane. In this case, as already mentioned before, we cannot expect to reach the running time of the pure floating-point version. Figure 14 shows our results for easy input data. and Figure 15 gives the running time for difficult input data.

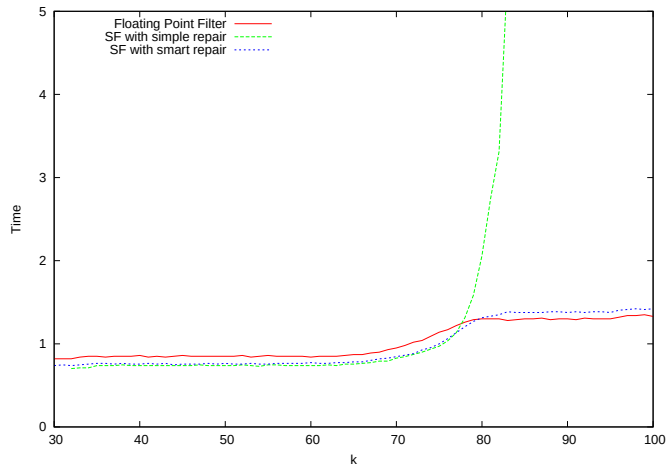

Figure 7: Range tree with difficult input.

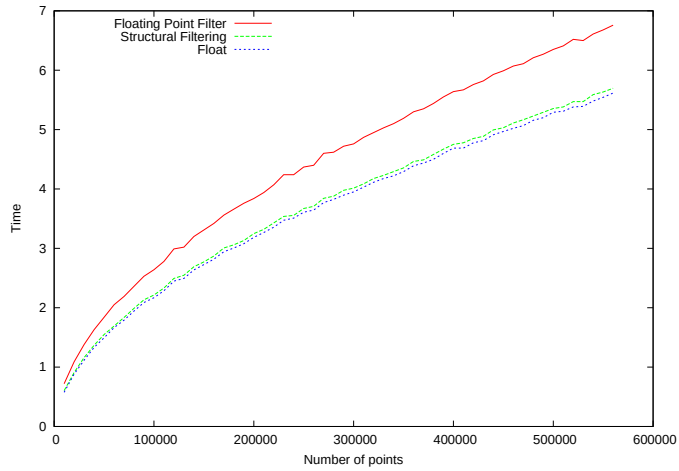

Figure 8: Point Location with easy input.

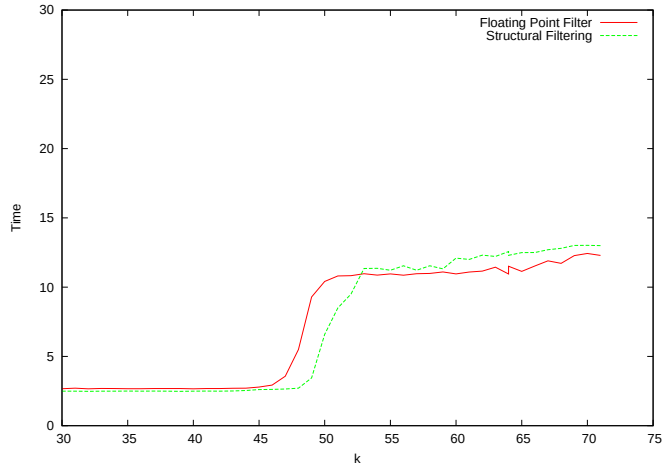

Figure 9: Point Location with difficult input.

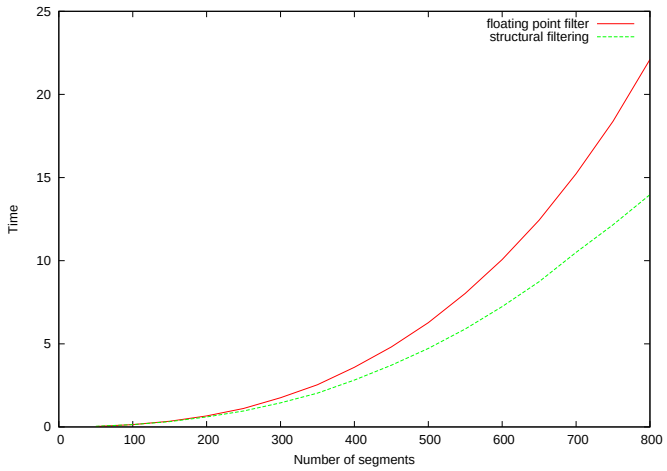

Figure 10: Sweep Segments with easy input.

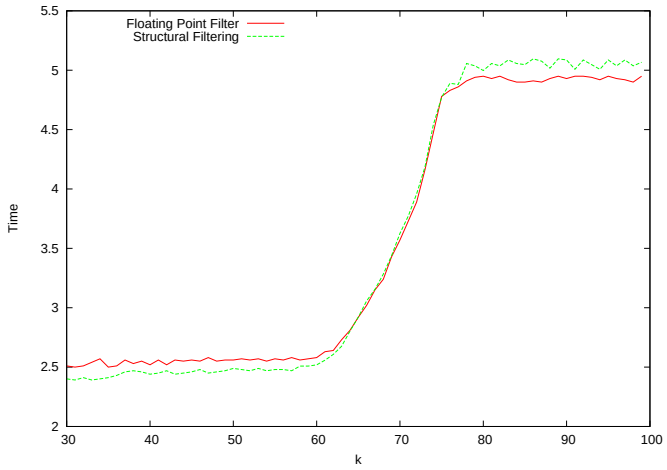

Figure 11: Sweep Segments with difficult input.

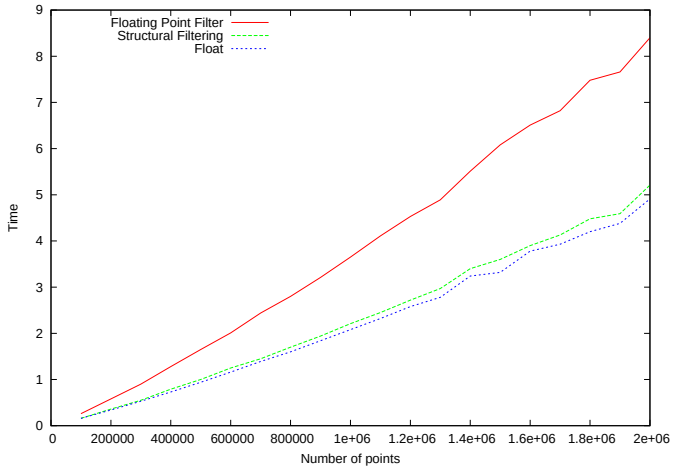

Figure 12: Convex Hull with easy input.

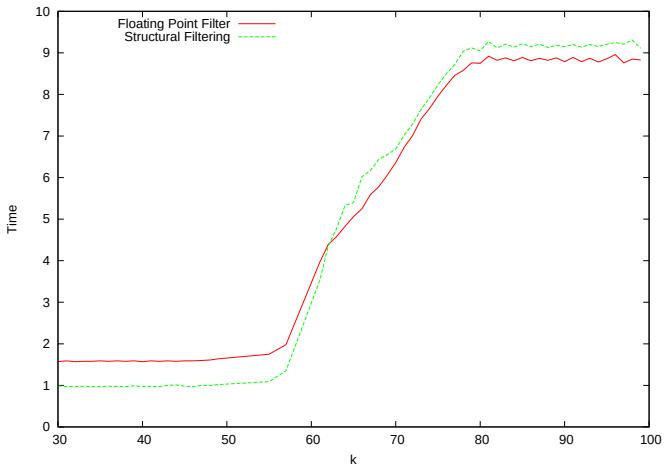

Figure 13: Convex Hull with difficult input.

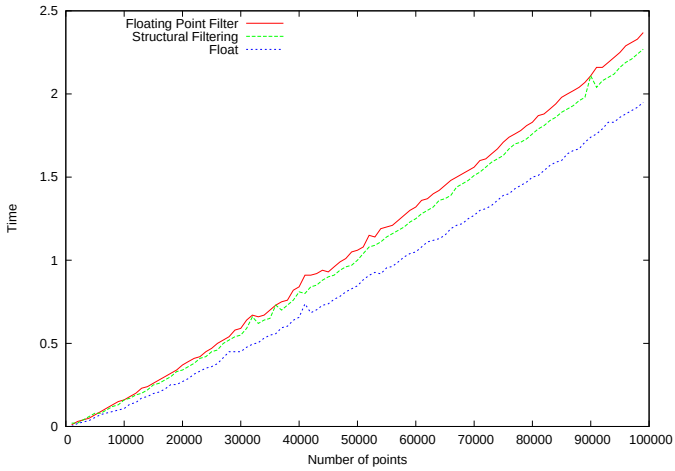

Figure 14: Delaunay Flipping with easy input.

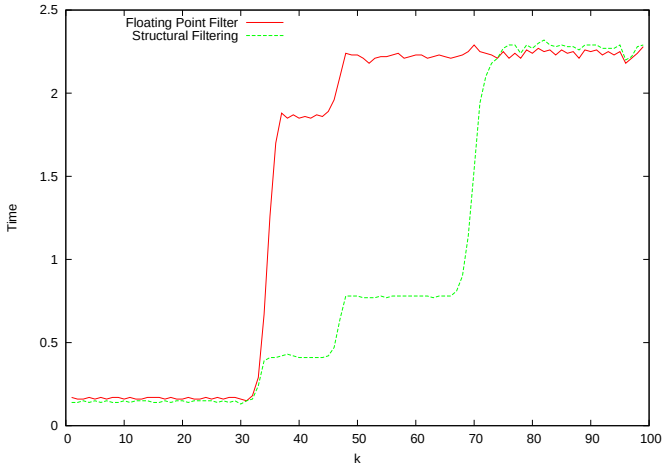

Figure 15: Delaunay Flipping with difficult input.

4 Conclusions

Structural filtering is a very powerful technique to speed up exact geometric computations. In this paper we showed that it is possible to achieve almost the same performance as in a pure floating-point version of the corresponding algorithm or data structure. Furthermore, one can avoid the bad behavior of simple repair procedures in presence of very difficult (i.e. high precision) input data by using clever repair strategies.

References

[1] J.L. Bently: "Multidimensional binary search trees" *Communications of the ACM*, 509-517, 1975

[2] Steven Fortune and Christopher J. Van Wyk: "Efficient Exact Arithmetic for Computational Geometry" *Proceedings of the 9th Annual Symposium on Computational Geometry*, San Diego, 163–172, 1993.

[3] Stefan Funke, Kurt Mehlhorn, and Stefan Näher. *Structural Filtering: a Paradigm for Efficient and Exact Geometric Programs.* Computational Geometry, Vol. 31(3), 179–194, 2005.

[4] L. Kettner, S. Näher: "Two Computational Geometry Libraries: LEDA and CGAL" *Handbook of Discrete and Computational Geometry*, Jacob E. Goodman and Joseph O'Rourke (Editors), Chapmann & Hall/CRC, 1435–1463, 2004

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

[6] S. Näher: "LEDA, a Platform for Cominatorial and Geometric Computing", *Handbook of Data Structures and Applications*, Dinesh P. Mehta and Sartaj Sahni (Editors), Chapmann & Hall/CRC, 41.1–41.18, 2004

[7] W. Pugh: "Skip Lists: A Probabilistic Alternative to Balanced Trees" *Communications of the ACM*, 668-676, 1990

[8] O. Devillers, S. Pion,M. Teillaud "Walking in a triangulation" *Annual Symposium on Computational Geometry*, 106-114, 2001

[9] C. L. Lawson "Transforming Triangulations" *Discrete Mathematics*, 365-372, 1972

[10] M. Kallay "The Complexity of Incremental Convex Hull Algorithms" *Info. Proc. Letters 19*, 197, 1984

[11] J. L. Bentley, T. Ottmann "Algorithms for reporting and counting geometric intersections" *IEEE Transactions on Computer Graphics Forum 15*, 205-217, 1979