# An Experimental Analysis of Floating-Point Versus Exact Arithmetic[*]

Martin Held[†] Willi Mann[†]

**Abstract**

In this paper we investigate how sophisticated floating-point codes that are in real-world use – VRONI for computing Voronoi diagrams, FIST for computing triangulations, and BONE for computing straight skeletons – can benefit from the use of the Core library (for exact geometric computing) or the MPFR library (for multi-precision arithmetic). We also discuss which changes to the codes were necessary in order to get them to run with these libraries. Furthermore, we compare our codes to codes provided by the CGAL project. By means of GMP-based (brute-force) verifiers we check the numerical validity of the outputs generated by all codes. As expected, the output precision of VRONI increases when MPFR is used, at a cost of an average slow-down by a multiplicative factor of 70. On the other hand, FIST demonstrates that a careful engineering can enable a code that uses floating-point arithmetic to run flawlessly, provided that the input coordinates are interpreted as genuine floating-point numbers. To our surprise, we could not get VRONI and BONE to work with CORE. It is similarly surprising that their CGAL counterparts did not fare well at all: we recorded drastically increased CPU-time consumptions combined with decreased accuracy of the numerical output.

## 1 Introduction

Robustness problems that occur for geometric codes when executed on a floating-point (fp) arithmetic are notorious. Typically, robustness problems are caused by numerical quantities being approximated, up to some quantitative error. While most errors tend to be benign, some errors may cause a program to end up in a state with no graceful exit, i.e., it crashes.

Various alternatives to standard floating-point computations have been advocated in recent years in an attempt to such robustness problems, such as the use of multi-precision arithmetic or exact geometric computing (EGC). The Core library, CORE [1], is an implementation of state-of-the-art EGC algorithms and techniques. It is written in C++, and was designed to be used easily as an alternative arithmetic back-end to existing C/C++ programs. CORE is a general-purpose tool that allows the correct evaluation of the sign of real predicates and, thus, is a way to ensure that the combinatorial part of an algorithm is computed exactly. The CGAL project [5] makes use of this EGC approach. Shewchuk [12] offers a small collection of geometric predicates that also support an exact evaluation based on standard fp-arithmetic. Another option is given by the MPFR library [3]: it is a “multi-precision floating-point library with correct rounding”, and can be considered as an intermediate step between IEEE 754 double-precision fp-arithmetic [10] and CORE.

Just how easy is it really to interface an existing geometric code with MPFR or CORE? And what do we gain or lose by resorting to MPFR, CORE or CGAL? In this paper we carry out an experimental case study that attempts to provide an answer to this question beyond personal beliefs or wide-spread myths. We consider three problems of imminent practical interest – the computation of triangulations, Voronoi diagrams and straight skeletons of polygons – and take three codes that compute these structures on a fp-arithmetic: the C codes FIST [6] and VRONI [7, 8], and the C++ code BONE [9]. All three codes were engineered to be reliable, and FIST and VRONI have been used extensively in industry and academia for more than a decade.

In Sec. 2 we discuss the modifications of our codes required to adapt them to a use with MPFR 3.0.1 or CORE 2.1, and report on problems encountered. (Due to lack of space we omit details on our results for BONE; they are similar to those for VRONI.) Section 3 documents the results of our run-time and verification tests.

## 2 Preparations

### 2.1 Modifications Required for CORE

CORE 2.1 data types do not work with C functions like `printf()` and `scanf()`, which we wanted to preserve in order to allow FIST to be compiled as a standard C program. For `scanf()` the problem can be worked around by implementing a custom version of `scanf()` that interprets the format specifiers for floating-point data types as specifiers for the CORE `Expr` type. Supporting `printf()` required more work because the arguments of a `printf()` command need to be converted to pointers. (Variable-argument functions cannot take C++ objects as arguments.)

Memory management also needed to be changed: The

[*]Work supported by Austrian FWF Grant L367-N15.

[†]Univ. Salzburg, FB Computerwissenschaften, 5020 Salzburg, Austria; {held,wmann}@cosy.sbg.ac.at

C functions `malloc()` and `free()` do not call constructors and destructors of C++ objects. We replaced them by the C++ operators `new` and `delete`.

And, of course, all precision thresholds used in comparisons of a numerical value with zero were set to zero. We were once again reminded of the fact that an $\varepsilon$-based comparison of some variable $x$ with zero should be encoded as "$|x| \leq \varepsilon$" rather than as "$|x| < \varepsilon$". (Otherwise, setting $\varepsilon := 0$ does not work.)

We also learned quickly that algorithmically equivalent code fragments may result in substantially different expression trees and, thus, runtimes for a CORE-based execution. For instance, in order to compute an (approximate) normal vector of a 3D facet whose vertices might be not perfectly co-planar it is advisable to first test whether the facet is truly planar. If yes then any three vertices that are not collinear allow to compute the correct normal vector. Otherwise, no correct normal vector exists and an approximate normal obtained by averaging normals defined by triples of vertices of the facet should be computed by using standard floating-point arithmetic, rather than by resorting to CORE and blowing up the size of the expression trees by making the normal dependent on all vertices of the facet.

### 2.2 Difficulties with CORE

Our work helped to reveal two major problems in CORE 2.0.8. The first problem concerns the parsing of the input: Some values in the interval $(-0.1, 0.1)$ were not parsed correctly. The CORE developers fixed the problem for positive numbers, and we extended the fix to negative numbers; it has become part of CORE 2.1.

Another problem is caused by the conversion from real to integer types. The CORE type for real numbers, `Expr`, supports a method called `intValue()`. The documentation shipped with CORE describes this method (and a few others) by the sentence "The semantics of these operations are clear." So we felt it safe to assume that `intValue()` behaves like the `(int)` conversion known from C. Unfortunately, it turned out the CORE conversion sometimes rounds up and sometimes rounds down. (It bases its rounding decision on a finite approximation of binary digits.) As a work-around we resorted to using the `floor()` function which works reliably on `Expr` numbers.

So far, we have been unable to execute CORE-based versions of VRONI or BONE on even trivial inputs. The apparent problem is that both codes cause large expression trees, where several minutes of CPU time do not suffice for CORE to evaluate the sign of one expression tree. (We created test cases and sent them to the developers of CORE, but no fix has been released yet.)

### 2.3 Adding MPFR Support

MPFR was not shipped with an integrated C++ wrapper, and the existing wrappers did not work with MPFR 3.0.1 when we tried them. (This has changed in the meantime.) So we wrote our own wrapper that supports precisely the operations needed in our applications, without adding any extra magic that might hurt the run-time performance.

As MPFR allows to set the precision at run-time, we have to adjust the precision thresholds used in the comparisons of fp-numbers. After some tests we ended up using the following formula[1]:

$$\varepsilon_{\text{prec}} := \varepsilon_{\text{fp}} / 2^{100 \cdot \left(\sqrt{\text{prec}/53} - 1\right)},$$

where $\varepsilon_{\text{fp}}$ is the standard threshold used for the fp-computations and prec is the precision (number of bits) requested. (The standard IEEE 754 precision assigns 53 bits of precision to the mantissa; see [10].)

We note that setting the default precision of MPFR in `main()` with the `mpfr_set_default_prec` library call does only affect variables created after this library call. In particular, the precision of global variables is not set adequately by default — and missing the correct setting of even just a few global variables turned out to downgrade the precision of the output quite significantly. So we modified our wrapper to ensure that the target variable in assignments has the default MPFR precision set.

### 2.4 Using Shewchuk's Predicates in FIST

Since correct sidedness tests are important for FIST we inserted a compile-time option that allows us to replace FIST's standard determinant evaluation by Shewchuk's 2D orientation predicate [12]. It was easy to integrate his code into FIST but one caveat remains: Shewchuk explicitly warns that his predicates will not work correctly if extended-precision registers are used. We circumvented this problem by running our tests on an x86-64 hardware; see the discussion in Sec. 3.2.

## 3 Tests and Experimental Results

### 3.1 Speeding up the CORE Library

The default constructor in CORE initializes an object that represents the constant zero. A closer inspection revealed that it always creates a new node representing zero. However, FIST very often uses variables in a way that causes their default constructor to be called: The programming language C (prior to the 1999 standard of C) forces variables to be declared prior to the body of the function, which often happens without an assignment in FIST. The default constructor is

[1]Thanks go to Stefan Huber for coming up with this formula.

also called frequently when arrays containing `Expr` objects are enlarged. A simple modification of this feature of CORE 2.0.8 resulted in an increase in speed of the CORE-based version of FIST by about 31%. (Our patch has been integrated into CORE 2.1.)

### 3.2 Influence of Compiler Options

In general, it is easier to debug non-optimized builds than optimized builds because compilers instructed to optimize may reorder instructions on machine-code level to gain performance. This often leads to a non-monotonic control flow with respect to the C sources. However, a key aspect of optimized builds is the attempt to keep variables inside registers across multiple statements in the source code.

On the x86 architecture, the floating-point registers are 80 bit wide, with a mantissa of 64 bit [11]. However, floating-point variables of type double as defined by IEEE 754-1985 [10] only have a mantissa of 53 bit. So, whenever a floating-point value is moved from a floating-point register to main memory, a loss of precision occurs. Or, in other words, the precision of numerical data computed depends on which variables and which intermediate results were kept in the registers. As a result, the output may change drastically once optimization is turned on.

Tests with FIST on our test data – see below – revealed that the outputs of the optimized build (using `gcc -O`) and the debug build (using `gcc -O0`) differ for about 15% of the inputs. We simply took the triangles (in the order computed by FIST) as the output. Hence, different outputs may nevertheless describe the same triangulation. Still, this means that at least one comparison in FIST returned different results on 15% of our inputs, depending on the compiler options.

There are multiple ways around this problem:

- Force the use of SSE instructions. This is not supported on all x86 CPUs. Many compilers including gcc use this as default on x86-64.
- Link the executable with a flag that limits the precision of the FPU. (E.g., `-mpc64` for gcc).
- Use a compiler flag that forces the write-back of all calculated values from the registers to main memory. (E.g., `-ffloat-store` for gcc.)

### 3.3 Test Results for Triangulations of Polygons

#### 3.3.1 Comparison of Different Arithmetic Backends

The following tests were conducted on the first author's set of polygonal areas which currently consists of 21 175 polygons with and without holes. The tests were run on an x86-64 hardware, an Intel Core i5 CPU 760 clocked at 2.80GHz. The test machine has 8 GiB of main memory, but virtual memory was limited to 6 GiB by the `ulimit` command. All codes were compiled with gcc 4.4.3.

We ran FIST with six different arithmetic back-ends:

- ordinary IEEE 754 double-precision fp-arithmetic (fistFp),
- Shewchuk's predicates (fistShew),
- CORE (fistCore),
- three precisions of MPFR: 53 bits (fistMp53), 212 bits (fistMp212), and 1000 bits (fistMp1000).

Figure 1 shows the run-time plots of fistFp, fistMp212, and fistCore. (The plots for the other three variants have been omitted due to lack of space.) The use of Shewchuk's exact predicates (fistShew) does not change the runtime behavior and results in a negligible speed penalty compared to fistFp: fistFp averages $0.155 \cdot n \log n$ microseconds, while fistShew averages $0.157 \cdot n \log n$ microseconds. All MPFR-based versions are about 24 times slower than fistFp on average, with fistMp212 averaging $3.786 \cdot n \log n$ microseconds, while fistCore is about 50 times slower. Increasing the precision requested for the MPFR-based versions causes no significant speed penalty. It is noteworthy, though, that the runtime of all MPFR-based versions varies much more significantly than for fistFp, fistShew, and fistCore.

In a second test we examined that numbers of data sets that were triangulated differently by the six variants of FIST. As it could be expected, there is no difference between fistFp and fistMp53. It is also not surprising that the use of exact predicates will cause some differences, despite the fact that considerable efforts were put into making FIST reliable. The difference between fistFp (fistMp53, resp.) and fistShew is small, though: Only 0.34 percent of the inputs were triangulated differently by fistShew. Contrary to our initial assumption, using MPFR with larger precisions in conjunction with FIST does not form an intermediate step between fistFp and fistCore: fistCore triangulates 10.38% of the inputs differently, while the outputs of fistMp212 and fistMp1000 deviate for 10.55% respectively 10.44% of the inputs.

#### 3.3.2 Verification of Triangulations Computed

Recall that different outputs need not indicate different or even incorrect triangulations. Since the number of differences was too large to be analyzed by hand, we wrote a code for verifying triangulations. A polygonal area is considered to be triangulated correctly only if the segments of the triangles neither intersect each other nor intersect the segments of the polygonal area. Note that we also consider triangulation edges that coincide (partially) with edges of the polygon or that pass through vertices of the polygon – termed "overlay" problems – as errors. Additionally, we check whether any segment of the triangulation passes outside of the polygon.

We use the Bentley-Ottmann algorithm [4] to find intersections. As arithmetic back-end, we use exact arithmetic based on the `mpq_t` data type provided by the GMP package (GMP 5.0.1, [2]). In an attempt to cut down the verification efforts, and lacking better means

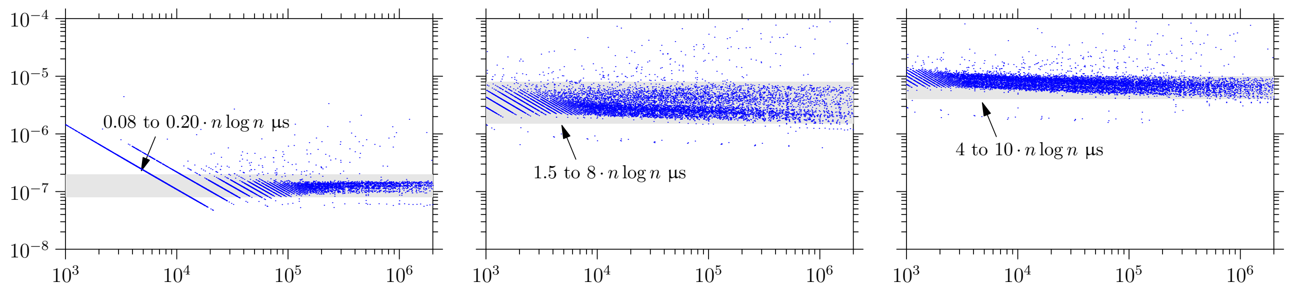

Figure 1: Run-time plots for fistFp, fistMp212, and fistCore. The $y$-axis corresponds to the run-time divided by $n \log n$, where $n$ is the number of vertices shown on the $x$-axis.

for establishing reference triangulations, we simply assume that all CORE-based outputs correspond to correct triangulations. Under this assumption it suffices to check all outputs that differ from outputs of fistCore.

In our first attempt to check the triangulations our GMP-based verifier took the input coordinates as exact values. (That is, 0.1 was regarded as $1/10$.) We were shocked to learn that the verifier reported about 4.92% of the triangulations to be faulty — uniformly for all variants of FIST which are not based on CORE. A more detailed analysis revealed that only fistFp, fistShew, and fistMp53 suffered from genuine intersection problems in their outputs, whereas the outputs of fistMp212 and fistMp1000 contained only overlay problems.

Still, nearly 5% faulty triangulations seemed too bad to be true. We implemented a viewer that does not suffer from floating-point errors and examined a few faulty triangulations: The errors reported by our verifier were only visible in the viewer when we used a zoom factor of at least $10^{15}$, which is approximately the reciprocal value of the precision of double precision fp-numbers.

So we switched to using the double-precision fp-approximations of the input coordinates as the true input numbers for our verifier, and again tested the triangulations reported to be faulty for fistFp: This test revealed no faulty triangulation at all. We conclude that the errors found with the first version of the verifier were only caused by input errors, i.e., by the small differences between real numbers and their fp-approximations. Since all real-world applications of FIST that we are aware of are based entirely on fp-numbers, triangulations computed by fistFp can rightfully be assumed to be correct from the point of view of these applications.

## 3.4 Test Results for Voronoi Diagrams of Segments

In the sequel, we report on tests of four variants of VRONI – VRONI based on fp-arithmetic (vroniFp), and VRONI based on MPFR with precisions 53, 212, 1000 (vroniMp53, vroniMp212, vroniMp1000) – and compare them to the Voronoi code shipped with CGAL 3.8.

We tested three variants of CGAL's Voronoi code. The first variant is fully based on double-precision fp-arithmetic (cgvdFp), the second variant uses `CGAL::Quotient<CGAL::MP_Float>` as predicate kernel (cgvdQu)and the third uses `CORE::Expr` as predicate kernel (cgvdEx). All variants use `Segment_Delaunay_graph_filtered_traits_2` template parameter to the underlying segment Delaunay graph class.

To ensure that CGAL and VRONI work on precisely the same input, we scale the input data to fit into the unit square as it is done per default in VRONI. For the same reason we add four dummy points outside of the bounding-box of the input explicitly to the input data for CGAL, at the positions specified by VRONI. (These points guarantee that each Voronoi cell of the actual input is bounded.) In order to ensure that the performance of CGAL is not influenced by file I/O we parse an input file and store the data in an intermediate data structure. Then we call the `insert()` method on the segment Delaunay graph, and construct the Voronoi diagram object based on the segment Delaunay graph. Only the insertions and the construction of the segment Delaunay graph are timed in our tests.

Our test bed consists of 18 787 input files with polygons, polygon areas, and polygonal chains. In order to ensure that all tests could be carried out within an acceptable time period we considered only inputs with at most 100 000 segments and limited the cpu-time consumption to 30 minutes per input. All tests were conducted on an Intel i7 CPU X 980 clocked at 3.33GHz. The test machine was equipped with 24 GiB of main memory.

The run-time performances of the variants tested are shown in Fig. 2. As expected, vroniFp is by far the fasted variant, averaging about $0.6 \cdot n \log n$ microseconds for inputs with 2 000 or more segments, while the MPFR-based variants all are 50–70 times slower. However, the variation of the run-time for different inputs of the same size is much smaller for the different variants of VRONI than for CGAL: Once the input size is large enough to make the timing reliable for VRONI there are

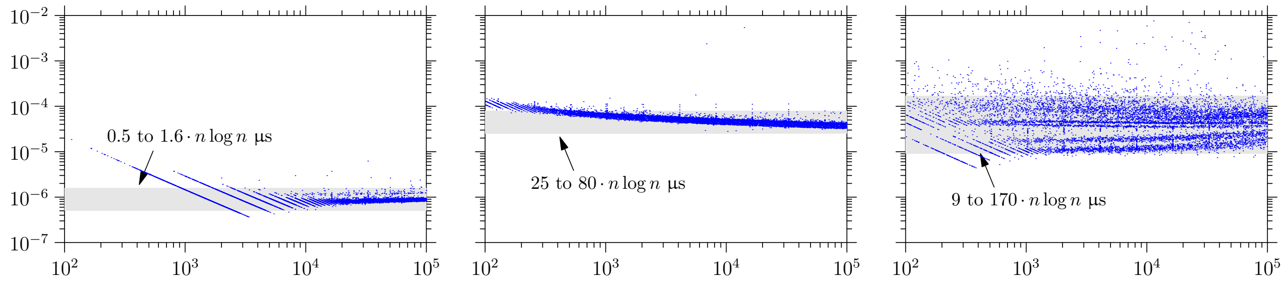

Figure 2: Run-time plots for vroniFp, vroniMp212, and cgvdEx. The $y$-axis corresponds to the run-time divided by $n \log n$, where $n$ is the number of vertices shown on the $x$-axis.

few data points outside of a small and dense band. On the contrary, the run-times of all CGAL variants vary by at least a multiplicative factor of 20. On average, CGAL performs slightly better than the MPFR-based variants of VRONI, but due to the big variance, there are several data sets that cause CGAL to perform worse than MPFR-based variants of VRONI. Interestingly, switching CGAL entirely to fp-arithmetic speeds up the code by only a factor of 1.5 on average. In any case, vroniFp completely outperforms CGAL on all data sets, being roughly 50–80 times faster than the CGAL variants. For about 0.36% of the inputs the CGAL variants did not finish within the limits imposed on runtime and memory.

The multiple outliers in the CGAL plots made us wonder whether there exist inputs for which the run-time complexity is worse than $O(n \log n)$. And, indeed, specific tests showed that smooth polygonal approximations of elliptical arcs or some free-form curves may cause all CGAL variants tested to consume $\Omega(n^2)$ time, with and without exact kernel, with and without filtered traits. (Unfiltered traits cause CGAL's built-in consistency checker to declare the solution as invalid quite frequently, which we also confirmed by our tests.)

Since the CGAL code was not designed to be run with conventional fp-arithmetic it is not surprising that cgvdFp fails frequently with fp-exceptions. (It crashed on 937 inputs.) It is more surprising, though, that the smallest input on which cgvdFp crashed also results in cgvdQu and cgvdEx computing Voronoi nodes which are numerically clearly wrong.

This observation made us wonder whether we had stumbled upon some isolated bug in CGAL, and we started to investigate the numerical accuracy of the Voronoi diagrams computed. We use an output format that stores the coordinates of the input sites and the coordinates of the Voronoi nodes calculated along with a reference to the defining sites. (We do not unscale the coordinates as this would introduce an unnecessary source of error.) We use a decimal precision of 60 digits in the output format. For each node, our verifier

- calculates differences in the distance to the defining sites of the node as the squared minimum distance divided by the squared maximum distance;
- checks whether any site $s$ is closer to the node than its closest defining site, and reports the squared distance to $s$ divided by the squared minimum distance.

In order to avoid introducing new errors in the verifier, all distance computations are based on GMP's `mpq_t` data type. Since brute-force all-pairs distance computations are used, we could afford to run our verifier only on comparatively small inputs with up to 2 000 segments.

Due to the use of fp-numbers in the output files of CGAL and VRONI we cannot expect any variant to have no error – but we can hope for small errors. The larger the error, the more such a ratio differs from one. For each variant all ratios different from one are sorted in increasing order and the square roots of the sorted ratios are subtracted from one. (These computations are based on standard fp-arithmetic.) In order to avoid plotting millions of points, we compute the averages of groups of 100 consecutive error values (in the sorted order) and plot the resulting errors.

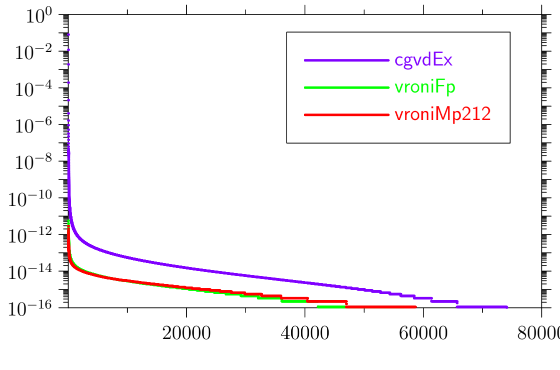

Figure 3: Error values that correspond to inconsistent distances to the defining sites of a node.

Figures 3+4 depict these sorted sequences of error values for vroniMp212 (lowest, red curve), vroniFp (middle, green curve), and cgvdQu (top, blue curve). Fig. 3 shows the errors that correspond to inconsistent distances to the defining sites of a node, and Fig. 4 shows the errors that represent violations of clearance disks by other sites.

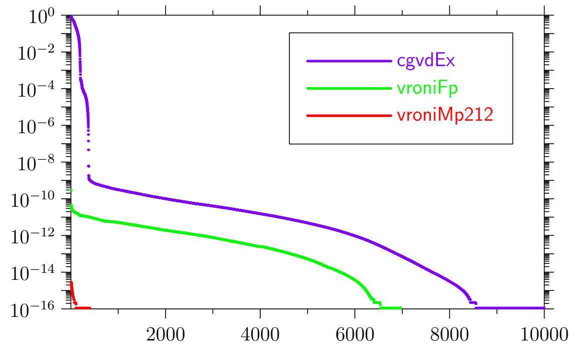

Figure 4: Error values that represent violations of clearance disks by other sites.

It is evident and not surprising that VRONI runs into numerical errors, and that using MPFR clearly helps VRONI to improve the accuracy of its output. But it is surprising to learn that all VRONI variants produce outputs which, on average, seem to be numerically much more accurate than any of the CGAL variants: vroniFp shows significantly fewer and smaller errors than cgvdQu. The numerical errors of cgvdEx are even more severe than those of cgvdQu, and cgvdFp is completely unreliable, given the large number of crashes observed.

Hence, if the numerical accuracy provided by the standard floating-point arithmetic is deemed insufficient in a real-world Voronoi application then it seems advisable to use VRONI in conjunction with MPFR, given the current state-of-the-art of Voronoi implementations. However, we note that the exact CGAL variants might determine correct Voronoi topologies even though the Voronoi nodes are less accurate than what can be achieved on a standard fp-arithmetic. (We had no means to assess and check this quality criterion.)

### 3.5 Test Results for Straight Skeletons

Due to lack of space we summarize our results for the computation of straight skeletons as follows: the MPFR-based versions of BONE are about 10–20 times slower than boneFp, with boneFp averaging about $30 \cdot n \log n$ microseconds on our test platform. CGAL's straight-skeleton code exhibits both a quadratic run-time as well as a quadratic memory consumption and, thus, is only feasible for very small inputs.

## 4 Conclusion

In this paper we discussed the problems that arose when we attempted to interface FIST, VRONI and BONE with the MPFR library or the CORE library. While MPFR was fairly easy to integrate into our C/C++ codes, the integration of CORE required significantly more efforts and non-trivial changes to the codes. Furthermore, the CORE-based version of FIST suffered from a substantial performance hit. Our tests suggest that the MPFR library should be considered if the numerical precision of a geometric code such as VRONI is of concern: It does indeed succeed in boosting the precision without causing the increase in runtime to become completely unbearable. As discussed, the numerical precision of the output of a geometric code may depend substantially on the compiler settings.

To our surprise, FIST run on a standard floating-point arithmetic performed flawlessy: all triangulations computed by FIST were correct. Also to our surprise, the CGAL alternatives to VRONI and BONE hardly are an alternative in practice; we observed a tremendously increased runtime (compared to VRONI and BONE) and a decreased numerical precision of the output.

Of course, our tests constitute case studies for only three specific applications with a small number of codes. Hence, generalizations of our findings need not be legitimate without probing the grounds. In particular, the mere fact that CGAL performed poorly in our test applications cannot be construed as an indication for a general weakness of CGAL for other applications.

## References

[1] CORE. `http://cs.nyu.edu/exact/core_pages/`.

[2] GMP. `http://gmplib.org/`.

[3] MPFR. `http://www.mpfr.org/`.

[4] J. Bentley and T. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[5] CGAL. `http://www.cgal.org/`.

[6] M. Held. FIST: Fast Industrial-Strength Triangulation of Polygons. *Algorithmica*, 30(4):563–596, Aug 2001.

[7] M. Held. VRONI: An Engineering Approach to the Reliable and Efficient Computation of Voronoi Diagrams of Points and Line Segments. *Comput. Geom. Theory and Appl.*, 18(2):95–123, Mar 2001.

[8] M. Held and S. Huber. Topology-Oriented Incremental Computation of Voronoi Diagrams of Circular Arcs and Straight-Line Segments. *Comput. Aided Design*, 41(5):327–338, May 2009.

[9] S. Huber and M. Held. Theoretical and Practical Results on Straight Skeletons of Planar Straight-Line Graphs. In *Proc. 27th Annu. ACM Sympos. Comput. Geom.*, pages 171–178.

[10] IEEE. *IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, 1985.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 1*, April 2011. `http://www.intel.com/products/processor/manuals/`.

[12] J. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete Comput. Geom.*, 18(3):305–363, Oct 1997.