Efficient Exact Geometric Computation Made Easy

C. Burnikel* R. Fleischer* K. Mehlhorn* S. Schirra*

Abstract

We show that the combination of the CGAL framework for geometric computation and the number type `leda_real` yields easy-to-write, correct and efficient geometric programs.

1 Introduction

The implementation of geometric algorithms is notoriously difficult. Geometric algorithms are usually designed for the so-called "Real RAM-model" which assumes exact real arithmetic (in the sense of mathematics). When real arithmetic is simply replaced by imprecise floating-point arithmetic, geometric algorithms that are provably correct with real arithmetic may crash or compute garbage. There are two ways to resolve this dilemma: one may either design new algorithms that work correctly even with imprecise arithmetic or implement the real RAM.

The number of problems that have been successfully attacked by the first approach is still small (see [13, 15, 28, 33] for surveys) and the techniques used in the solutions do not obviously generalize to other geometric problems. The main disadvantage of the approach is the necessary redesign; the approach requires to redo computational geometry.

The alternative is to base geometric computation on the Real RAM model [14, 17, 20, 32, 34]. This is attractive because it allows one to use the algorithms and data structures developed under the Real RAM assumption without redesign. We show that the combination of the CGAL [3, 9, 12, 25, 31] framework for geometric computation with the number type `leda_real` of LEDA [21, 22] provides an easy to use and efficient basis for exact geometric computation.

CGAL is a very flexible and convenient basis for geometric programming: Algorithms and data structures in CGAL are generic. They are parameterized by the types used by them, in particular, by the basic geometric objects (like points, lines, segments) and primitives (like orientation predicate) provided by a geometry kernel. The algorithms and data structures work with *any* implementation of these types as long as they fulfill certain syntactic and semantic requirements. The algorithms are correct as long as the primitives are correct. In particular, combining a CGAL-algorithm with any exact geometry kernel yields a correct program. The user may choose the geometry kernel according to her needs.

CGAL offers two geometry kernels itself, one based on the representation by Cartesian coordinates and one based on the representation by homogeneous coordinates [11]. Both kernels are parameterized with the number type used to store coordinates and to do calculations. The use of an exact number type yields an exact kernel. The number type may be chosen according to the requirements of the application. For example, for computations inside the rational numbers, an arbitrary precision rational number type could be used with the Cartesian kernel or alternatively, an arbitrary precision integer type with the homogeneous kernel. The rational geometry kernel of LEDA also supports exact computations in the rational domain.

For computations that require algebraic numbers the number type `leda_real` can be used; it provides exact computation with addition, subtraction, multiplication, division, k-th root operations, and comparisons. We present this number type and describe its implementation in Section 2.

*Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany, Email: {`burnikel, rudolf, mehlhorn, stschirr`}`@mpi-sb.mpg.de`. This work was partially supported by ESPRIT LTR projects 21957 (CGAL) and 28155 (GALIA)

Alternatives to leda_reals are discussed in Section 3.

In Section 4 we discuss the combined use of CGAL and leda_reals in four experiments and compare them to other approaches to exact computation. We will see that CGAL is easily combined with leda_reals and other methods for exact geometric computation and that leda_reals are an efficient basis for exact computation.

2 The Number Type leda_real

The number type leda_real is a C++ class for exact computation with numbers that can be constructed by the algebraic operations $+$, $-$, $\cdot$, $/$, and $\sqrt[k]{\ }$ from the integers. In other words, every integer is a leda_real, and the type is closed under the operations $+$, $-$, $\cdot$, $/$, and $\sqrt[k]{\ }$. All comparison operations are exact.

The leda_reals are as easy to use as any of the built-in number types of C++. To illustrate the point, we show how to compute the sign of $E = \sqrt{9+4\sqrt{2}} - (1+2\sqrt{2})$.

```
leda_real root_two = sqrt(leda_real(2));
leda_real E = sqrt(9 + 4 * root_two)
              - (1 + 2 * root_two);
cout << sign(E);       // prints 0
```

The output statement in the last line prints 0; observe that the expression E is a complicated way of writing the number zero. When the same chunk is evaluated with doubles instead of leda_reals, E evaluates to $-4.44 \cdot 10^{-16}$ and consequently the output statement prints -1.

We next discuss the key features of the implementation of leda_reals. The detailed implementation can be found in [8].

A leda_real is represented as a directed acyclic graph (dag) that records its construction. The nodes of the dag for an expression E correspond to the subexpressions of E. The leaves of the dag are integers, and each internal node of the dag is labeled with a unary or binary operation and points to the operands from which the subexpression was computed. Fig. 1 shows the dag for the expression E of our first example.

Arithmetic operations are constant time operations. An arithmetic operation simply constructs a new node in the graph representation, establishes pointers to the subexpressions, labels the node with the type of the arithmetic operation, and computes an initial approximation by executing the arithmetic operation with double precision floating-point arithmetic.

Comparisons are reduced to sign computations. The sign of a leda_real E is determined by

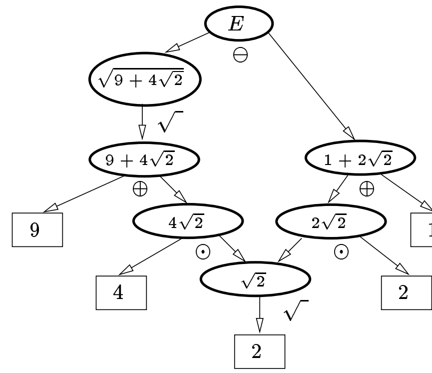

Figure 1: The expression dag for $E = \sqrt{9+4\sqrt{2}} - (1 + 2\sqrt{2})$

numerical computation. We numerically compute approximations $\widetilde{E}$ of E with increasing quality until either the sign of E is equal to the sign of $\widetilde{E}$ or until a separation bound allows us to conclude that E is zero. A separation bound for an expression E is a number sep_E such that

$$|\,E\,| < sep_E \text{ implies } E = 0.$$

We use the easily computable separation bound presented in [5]. We briefly review the method for the important special case of a *division-free* expression E. Let ξ denote the exact value of E, let $p_\xi(x)$ be the minimal polynomial of ξ and and let d be the degree of $p_\xi(x)$. By definition, ξ is a root of $p_\xi(x)$. The other roots $\xi_2, \ldots, \xi_d$ are called the *conjugates* of $\xi = \xi_1$. Since E is division-free, the polynomial p_ξ has the representation $p_\xi = \prod_{i=1}^{d}(x - \xi_i)$. In particular, the product of all the ξ_i is the constant term c of p_ξ. If ξ is nonzero, c is a nonzero integer. Now if we know an upper bound U for all $|\,\xi_i\,|$, we find that

$$|\,\xi\,| = |\,c\,| \cdot \prod_{i=2}^{d} |\,\xi_i\,|^{-1} \geq U^{1-d}$$

and this will be our *separation bound.*

Note that for computing a separation bound we don't need the minimal polynomial itself but only bounds for its degree and for its roots. Bounding the degree is simple: If E contains n roots of orders $d_1, \ldots, d_n$ we have $d \leq \prod_{i=1}^{n} d_i$. Finding an appropriate U is easy as well. Namely, if we replace in the expression E all integers by their absolute values and all subtractions by additions, the value of the resulting expression E^+ is an upper bound for the $|\,\xi_i\,|$. The exact computation of E^+ is difficult, but is suffices to compute an upper bound for it. Actually, we compute an upper bound for $\log E^+$ using double precision floating-point arithmetic. This concludes the case of division–free expressions. The general case of expressions involving divisions can be reduced to the special case. The resulting separation bounds are weaker, see [5].

The following pseudo-code summarizes the sign computation.

prec $= 52$;
compute U and d;
do {
  error $= U \cdot 2^{-\text{prec}}$;
  compute an approximation $\widetilde{E}$ of E
    such that $|E - \widetilde{E}| \leq$ error;
  if ($|\widetilde{E}| >$ error) return $sign(\widetilde{E})$;
  prec $= 2 \times$ prec;
} **while** (error $\geq U^{1-d}/2$);
return 0;

We will explain below how the approximation $\widetilde{E}$ of E is computed. The sign computation stops when either $|\widetilde{E}| >$ error or error $< U^{1-d}/2$. The latter will be the case if $E = 0$ and the former will be the case if $E \neq 0$. Moreover, for nonzero E the approximation with half the final precision did not suffice to determine the sign of E and hence the precision used in the last iteration is at most twice the optimal precision. Because the precision is doubled in every iteration, the running time of the sign test is dominated by the last evaluation. Our scheme is *adaptive*; easy sign computations are fast, and difficult sign computations are more expensive. Fig. 2 in Section 4 illustrates the adaptiveness of our scheme. The values of the expressions E_1 and E_2 are well separated from zero and hence a low–precision–computation determines their sign; E_3 is zero and hence the full precision dictated by the separation bound is needed to compute its sign.

We use *precision-driven computation* (see also [24, 34]) to compute an approximation $\widetilde{E}$ of an expression E with prescribed error. Precision-driven evaluation is a recursive procedure with two arguments, an expression E and an (absolute) error bound B. It computes error bounds B_1 and B_2 for the subexpressions E_1 and E_2 of E (only B_1 in the case of a unary operation) and a mantissa length L such that the computation of approximations $\widetilde{E_1}$ and $\widetilde{E_2}$ with error bounds B_1 and B_2, respectively, and the computation of $\widetilde{E}$ from $\widetilde{E_1}$ and $\widetilde{E_2}$ by floating-point arithmetic with mantissa length L guarantees an error B for $\widetilde{E}$. We use the number type `leda_bigfloat` for it; it supports floating-point arithmetic with arbitrary mantissa length. Next, we explain precision driven computation in more detail for the case of a multiplication, for the other operations we refer the reader to [8] .

Consider a multiplication $x \cdot y$. Assume that $\hat{x}$ and $\hat{y}$ are approximations of x and y, respectively, and that ϵ is the relative error of the floating-point realization of the multiplication. Let $q_x = |x - \hat{x}|$ and $q_y = |y - \hat{y}|$ be the errors in the approximations of x and y, respectively. We have

$$\begin{aligned} &|\widehat{x \cdot y} - (x \cdot y)| \\ &\quad \leq \ |\widehat{x \cdot y} - (\hat{x} \cdot \hat{y})| + |\hat{x} \cdot (\hat{y} - y)| + |y \cdot (\hat{x} - x)| \\ &\quad \leq \ |\widehat{x \cdot y}| \cdot \epsilon + |\hat{x}| \cdot q_y + |y| \cdot q_x. \qquad (1) \end{aligned}$$

Let q be the error bound that we want to achieve for the result of the multiplication $x \cdot y$. A sufficient condition to guarantee the maximal error q is to have the first term of relation (1) bounded by $q/2$ and the two other terms by $q/4$:

$$\epsilon \leq \frac{q}{2|\widehat{x \cdot y}|}\ ,\ q_y \leq \frac{q}{4|\hat{x}|}\ ,\ q_x \leq \frac{q}{4|y|}. \qquad (2)$$

We first deal with the constraint on q_x. Not knowing the exact value of y, we use its already computed upper bound y^+ and request that x be approximated with the error bound $q/(4|y^+|)$. The resulting approximation $\hat{x}$ is then used to compute an admissible error bound q_y and we request the approximation of y with the error bound q_y. To fulfill the constraint on the relative precision ϵ of the multiplication $\hat{x} \cdot \hat{y}$ we take a rough upper bound M for its yet unknown result, e.g., a bound that depends only on the binary exponents of $\hat{x}$ and $\hat{y}$. Then we chose $\epsilon < q/(2M)$ and perform the operation in that precision. This concludes the description for multiplication. The remaining operations $+, -, /, \sqrt[k]{\ }$ are handled similarly.

A first version of `leda_reals` was presented in [7]. In the first version we used a weak separation bound. In [5] we proved a much better separation bound. We modified the implementation to work with the new separation bound and introduced k-th roots for $k > 2$.

3 Alternative Approaches

In this section we discuss other approaches to exact computation. We categorize them by how they represent algebraic numbers. The `leda_reals` use an expression dag representation.

The package which comes closest to `leda_real` is the `Real/Expr`-package [24, 34]. The package offers a similar user interface and also uses a similar implementation strategy using expression dags. The differences are: (1) only square roots are supported, (2) a weaker separation bound is used, and (3) precision-driven computation is implemented differently: Each node of the dag representing an expression has two error bounds associated with it: a bound on the relative error and a bound on the absolute error. The evaluation strategy guarantees that at least one of the bounds is satisfied and the bounds are chosen such that satisfaction of one of the bounds allows the computation of the sign. The bounds are determined top-down. The more recent CORE-package [16] uses the `Real/Expr`-package as its computational engine and hence shows the same running time behavior. In our experiments, we used the version of the `Real/Expr`-package coming with CORE 1.1.[1] The `Real/Expr`-package uses the number type `Integer` that comes with the `g++` compiler. In our tests, the `Real/Expr`-package was less efficient than the `leda_reals` (see Section 4).

The method described in [23] represents the value of an expression involving roots of (algebraically independent) numbers n_1, n_2, $\ldots$, n_k as an element of the extension field $\mathbb{Q}(n_1)(n_2)\ldots(n_k)$; only square roots are supported. The representation is unique and hence arithmetic operations are costly; zero tests are cheap (as the number zero has a unique representation). Sign tests are implemented by a combination of numerical and symbolic computation. A low precision numerical approximation is computed first; if it does not determine the sign, the representation in the quadratic extension field is used for sign determination. An implementation was made in Scheme, but it is not publicly available.

Any algebraic number can also be represented as the root of a polynomial; roots can be specified either by isolating intervals or by the sign sequence of all derivatives of the polynomials at the root. The polynomial may either be an arbitrary polynomial having the algebraic number as one of its roots or a minimal polynomial. With this representation each arithmetic operation requires the computation of a new polynomial having the result of the arithmetic operation as one of its roots. Such polynomial can be computed by resultant computations. The required algebraic computations are available in computer algebra systems[2] and algebra toolkits, e.g. [27] and [19]. We asked the MAPC-group [19] to compute the sign of the expression E_3 in Fig. 2 for $L = 1000$; the computation took more than 50 seconds [18].

The main advantage of the expression dag methods seems to be that arithmetic operations have cost $O(1)$ and that the cost of a sign computation is related to its difficulty. In contrast, the other methods need to compute an algebraic representation for every intermediate result and the non-constant cost of this is not directly related to the difficulty of a later sign computation.

A polynomial-time probabilistic zero-test for sums of radicals was recently given by Blömer [2]. His method works for sums of arbitrary roots (nested roots are not allowed) and thus only for a subset of the expressions handled by `leda_reals`. The method is probabilistic and hence may err with small probability (a non-zero number may be declared zero). The running time is polynomial in the size of the expression. The method could be used as an alternative stopping criterion in the numerical computation of signs. This would not speed up the sign determination of numbers which are very close to zero, but would probably speed up the sign determination of numbers which are zero. Blömer mentions preliminary experiments in his paper, but no implementation is publicly available.

4 Experiments

We performed four experiments: (1) The computation of the signs of expressions $E_1 = 1 + 2\sqrt{x}$, $E_2 = \sqrt{4x + 1 + 4\sqrt{x}}$, and $E_3 = E_2 - E_1$ for a random number x with L decimal places. (2) The computation of the convex hull of intersections of line segments. (3) The computation of the convex hull of intersections of circles. (4) The computation of single source shortest paths in graphs with Euclidean edge lengths.

The second experiment stays within the rational numbers. We used it to compare `leda_reals` with packages intended for exact and efficient computation with rational numbers. The third and the last experiment require algebraic numbers; the algebraic numbers in the third experiment have bounded expression size and the algebraic numbers in the last experiment are sums of an arbitrary number of $\sqrt{2}$ s.

All experiments were run on a SUN SPARC Ultra 10 with 333 Mhz processor and 128 MB main memory. All running times are in seconds. The complete code of all experiments is available on the net [4].

4.1 A First Simple Example

We generated a random integer x with L decimal digits and then computed the signs of the expressions $E_1 = 1 + 2\sqrt{x}$, $E_2 = \sqrt{4x + 1 + 4\sqrt{x}}$, and $E_3 = E_2 - E_1$. Observe that the sign of E_3 is zero and that the sign of the other two expressions is positive. Fig. 2 shows the running times of

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

[2] see for example Computer Algebra Information Network at `http://www.riaca.win.tue.nl/CAN/`.

L	1000	2000	4000
$E_1 = 1 + 2\sqrt{x}$	≤ 0.01	≤ 0.01	≤ 0.01
$E_2 = \sqrt{4x + 1 + 4\sqrt{x}}$	≤ 0.01	≤ 0.01	≤ 0.01
$E_3 = E_2 - E_1$	0.57	2.01	7.35

Figure 2: Running times of the first experiment. We generated a random integer x with L decimal digits and computed the sign of the expressions shown. The sign of E_3 is zero, and the sign of the other two expressions is one. The running time of the sign computations is shown.

`leda_real`s for different values of L. In comparison, the `Real/Expr`-package could not compute the sign of E_3 for $L = 1000$ within an hour.

4.2 Convex Hulls of Segment Intersections

In this experiment, we computed the convex hull of the intersection points of line segments. We used the CGAL implementation of the throw-away algorithm [1] for computing planar convex hulls. This algorithm is parameterized with a point class which must provide two predicates: a predicate to compare points lexicographically by their Cartesian coordinates and a predicate to check whether three points form a left-turn. The algorithm can be used with any point class that provides these predicates. We used the point classes provided in the CGAL kernel templates and in the rational geometry kernel of LEDA.

CGAL [11] provides two geometry kernel templates. The Cartesian geometry kernel template is parameterized with a number type `NumberType` and represents points with Cartesian coordinates in `NumberType`. The number type must support divisions. In our experiments, we used the built-in number type `double`, the LEDA number type `leda_real`, the LEDA number type `leda_rational` for exact computation with rational numbers of arbitrary size, and the number type `Expr` from the `Real/Expr`-package.

The homogeneous geometry kernel template is also parameterized with a number type and represents all points with homogeneous coordinates[3] in this number type. The number type does not need to support divisions. In our experiments, we used the number types `Expr`, `leda_real`, `leda_bigfloat` (in the exact[4] mode), the LEDA number type `leda_integer` for exact computation with integers of arbitrary length, and the CGAL number type `CGAL_Expanded_double`[5].

The rational geometry kernel of LEDA provides points with rational coordinates and all predicates are evaluated exactly. We refer to this kernel as the `rat_leda` kernel. The kernel is known to be very efficient (see [29] and the chapters on Geometry Kernels and Geometry Algorithms of [21]). Points are represented by their homogeneous coordinates. The kernel uses a *floating-point filter* [17, 14, 20, 6] to speed up the evaluation of geometric predicates. For example, in order to compute the orientation of three points the determinant shown below is first approximately evaluated with floating-point arithmetic. If the approximation is sufficiently far away from zero so as to guarantee that the exact value of the determinant has the same sign, the sign of the approximation is returned as the result of the orientation predicate. If not, the value of the determinant is computed exactly using `leda_integer` and its sign is returned. The details are described in the chapter on Geometry Kernels of [21]. The efficacy of the floating-point filter depends on the size of the coordinates.

We performed two variants of the experiment. In the first variant the segment endpoints had integer coordinates (type `int`) and were chosen to lie near the circle centered at the origin and having radius 2^{bitsize}. We used bitsize $= 30$ for Fig. 4 and had the bitsize vary in Fig. 5. In the second variant the segments endpoints had floating-point coordinates of type `double` and were chosen to lie near the circle centered at the origin and having radius 250.

Fig. 3 shows the (almost) complete program for the second variant of the experiment for Cartesian kernels with number types `double` and `leda_real`, respectively. Only one line has to be added to perform the experiment with another number type and Cartesian coordinates.

For points with `double` coordinates we used two ways to transform Cartesian coordinates (x, y) of type `double` into integer homogeneous coordinates. We have $x = X/W_x$ and $y = Y/W_y$ for integers X, Y, W_x, W_y, where W_x and W_y are powers of two. Let $W = \max(W_x, W_y)$ and $x' = xW/W_x, y' = yW/W_y$. In the first method `(1)` we used the homogeneous coordinates (x', y', W), and in the second method `(2)` we converted x and y to rational numbers of type `leda_rational` and then constructed the homogeneous coordinates from the rational numbers; the implementation of this conversion process in LEDA leads to homogeneous coordinates (xW_y, yW_x, W_xW_y). The second method leads to much larger integers. In Fig. 6 we refer to the two methods by

[3] A point with homogeneous coordinates (X, Y, W) has Cartesian coordinates $(X/W, Y/W)$; W is called the homogenizing coordinate.

[4] In the exact mode, `leda_bigfloat`s incur no rounding error but adapt the mantissa length automatically (except for division operations).

[5] `CGAL_Expanded_double` represents numbers exactly by sums of `double`s as described by Dekker [10] and later refined by Priest [26] and Shewchuk [30].

The input to the following function is a set of segments with Cartesian `double` *coordinates. The input is passed to the function as the sequence of segments starting with the segment pointed to by the iterator* `first` *and ending with the segment pointed to by the predessor of iterator* `last`*. The function performs the experiment with kernel* `Kernel`*. The function object* `converter` *converts segments with* `double` *coordinates to the segment type of* `Kernel`*.*

```
template <class InputIterator, class Kernel, class SegmentConverter>
float compute( InputIterator first, InputIterator last,
               const Kernel&, const SegmentConverter& converter )
{
  // typedefs for Kernel types
  typedef CGAL_Point_2<Kernel>      Point;
  typedef CGAL_Segment_2<Kernel>    Segment;

  // convert segments with double coordinates to segment type of Kernel
  vector&lt;Segment&gt;  segments;
  transform( first, last, back_inserter( segments), converter);

  // point containers for intersection points and convex hull
  vector<Point >   intersections;
  vector<Point >   convex_hull;


  // measure time for intersection and hull computation with Kernel types
  CGAL_Timer  watch;
  watch.start();
  CGAL_segment_intersection_points_2( segments.begin(), segments.end(), back_inserter( intersections));
  CGAL_convex_hull_points_2( intersections.begin(), intersections.end(), back_inserter( convex_hull));
  watch.stop();
  return watch.time();
}
```

The following function object class performs the conversion from segments of the CGAL kernel with Cartesian representation with `double` *coordinates to Cartesian representation with* `NumberType` *coordinates.*

```
template <class NumberType>
struct C_double_to_C
{
  typedef CGAL_Point_2< CGAL_Cartesian< NumberType> >    Point;
  typedef CGAL_Segment_2< CGAL_Cartesian< NumberType> >  Segment;

  Segment operator()(  const CGAL_Segment_2< CGAL_Cartesian< double> >& s) {
            return Segment( Point( NumberType(s.source().x()), NumberType(s.source().y()) ),
                            Point( NumberType(s.target().x()), NumberType(s.target().y()) );
          }
};
```

The following lines perform the experiment for Cartesian kernels with number types `double` *and* `leda_real`*. Here,* `S` *is assumed to be a vector of segments with* `double` *coordinates. The code to fill* `S` *is not shown.*

```
  cout << "C<double> ";
  cout << compute(S.begin(), S.end(), CGAL_Cartesian<double>(), C_double_to_C<double>()) << endl;
  cout << "C<leda_real> ";
  cout << compute(S.begin(), S.end(), CGAL_Cartesian<leda_real>(), C_double_to_C<leda_real>()) << endl;
```

It is very easy to extend the experiment to a Cartesian kernel with another number type, e.g. `Expr`*:*

```
  cout << "C<Expr> ";
  cout << compute(S.begin(), S.end(), CGAL_Cartesian<Expr>(), C_double_to_C<Expr>()) << endl;
```

Figure 3: The program for computing the convex hull of intersection points of line segment.

C<double>	0.01
rat_leda	0.02
H<leda_integer>	0.18
C<leda_real>	0.25
C<leda_rational>	0.27
H<leda_real>	0.31
H<Expanded_double>	0.33
H<leda_bigfloat>	0.62
C<Expr>	5.89
H<Expr>	8.04

Figure 4: Running times for the convex hull of the intersection points of 60 line segments with endpoints with integral coordinates of at most 30 bits using different kernels. C stands for the Cartesian CGAL kernel template, H stands for the homogeneous CGAL kernel template, rat_leda stands for the rational geometry kernel of LEDA. Note that computations with the number type double are not guaranteed to be correct.

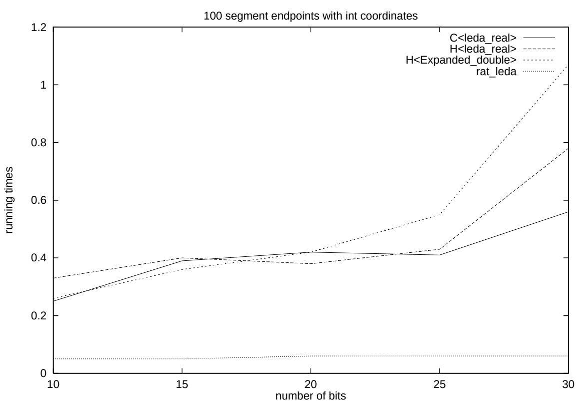

Figure 5: Running times for the convex hull of line segment intersection points. The segment endpoints had b-bit integer coordinates. The figure shows the dependence of the running time on b for four different kernels.

the qualifiers (1) and (2). For the conversion to the homogeneous CGAL kernel with number type leda_integer we used the first method, too.

The problem stays within the rational domain, but it is computationally demanding. Consider the case that the endpoints have b-bit integer Cartesian coordinates. Then the Cartesian coordinates of the intersection points are rational numbers whose numerator has up to $3b + O(1)$ bits and whose denominator has up to $2b + O(1)$ bits. The orientation predicate for three points p, q, and r requires to compute the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 \\ p_x & q_x & r_x \\ p_y & q_y & r_y \end{vmatrix}$$

If p, q, and r are intersection points, the determinant is the sum of three rational numbers whose numerator has up to $5b + O(1)$ bits and whose denominator has up to $4b + O(1)$ bits. The value of the determinant is therefore a rational number whose numerator has up to $13b + O(1)$ bits and whose denominator has up to $12b + O(1)$ bits. For $b = 30$ this amounts to about 390 and 360 bits, respectively. Given the required computational demand it is not surprising that the computation with doubles was not always correct. Frequently non-extreme points were reported to be extreme. All other kernels and number types guarantee an exact result.

C<double>	0.01
rat_leda (1)	0.04
C<leda_real>	0.63
H<leda_real>	0.68
H<leda_integer>	0.74
H<Expanded_double>	1.06
rat_leda (2)	1.36
H<leda_bigfloat>	1.45
C<Expr>	8.87
H<Expr>	10.14
C<leda_rational>	19.29

Figure 6: Running times for the convex hull of the intersection points of 60 line segments with endpoints with double coordinates using different kernels. The qualifiers (1) and (2) are explained in the text. Note that computations with the number type double are not guaranteed to be correct.

We next explain the numbers in Figures 4, 5, and 6. The computation with doubles is clearly the fastest. Among the exact kernels the rational geometry kernel of LEDA is by far the fastest. leda_reals turned out to be slower by a factor of 12 for the case of integer coordinates and a factor of 15 for the case of double coordinates. They outperformed Expr by a wide margin. leda_reals do very well in comparison to the number types leda_bigfloat, leda_integer, leda_rational, and CGAL_Expanded_double. All four number types use no floating-point filter and hence the computation of an orientation predicate forces the exact computation of the value of the orientation determinant, an expensive task.

There is an essential difference between the floating-point filters used in leda_real and Expr and the floating-point filter used in rat_leda. The latter uses a double-computation for the filter stage and hence the filter becomes ineffective once the input precision is above a certain threshold. This is clearly shown by the dramatically different running time for the two methods of converting double Cartesian coordinates to homogeneous integer coordinates of rat_leda. Recall that method (2) leads to larger integer coordinates. The floating-point filters in leda_real and Expr use floating-point arithmetic with arbitrary

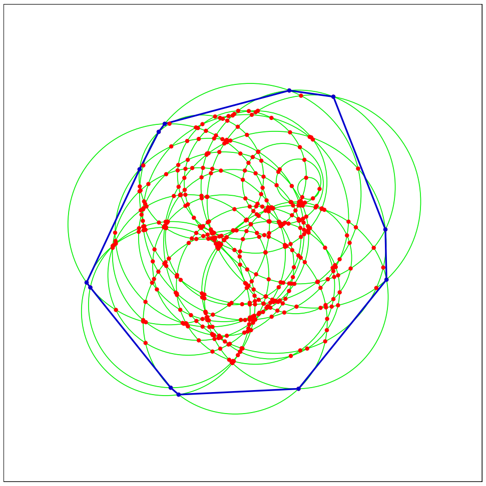

Figure 7: Convex hull of intersection points of circles.

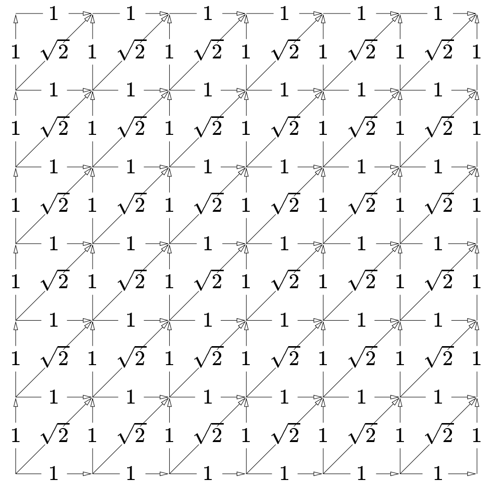

Figure 8: $n \times n$ grid graph with diagonals for $n = 7$.

mantissa length and hence the floating-point filter becomes slower for higher precision inputs but it never becomes completely ineffective. This is visible in Fig. 5.

For number types supporting division (such as `leda_real`) one has the choice between using the Cartesian and the homogeneous kernel of CGAL. We were surprised to see that there seems to be little difference between the two kernels; we have no quantitative explanation. There are two effects which certainly play a role: (1) The Cartesian representation requires fewer arithmetic operations to perform a test. (2) The Cartesian representation uses division and our separation bound is weaker for expressions involving divisions. Observe that there are many collinear points in our input set and hence many difficult orientation tests. The relative merits of the two representations deserve further study.

4.3 Convex Hulls of Circle Intersection points

We turn to our third experiment, the computation of the convex hull of intersection points of circles. This experiment requires the use of algebraic numbers. The expressions defining the algebraic numbers have bounded size. The midpoints of the circles had integral coordinates of at most 16 bits chosen at random. The square of the radius was an integer with at most 32 bit, chosen at random. Fig. 7 shows an example.

The `double`-computation was about 15 to 20 times faster than the computation with `leda_reals`, which in turn was about 50 times faster than `Expr`. The latter difference is explained by the better separation bound used in `leda_reals`. The former difference is surprisingly small and is about the same as in the experiment of the preceding section.

4.4 Shortest Paths

Our last experiment uses algebraic numbers where the defining expressions do not have bounded size. We considered the single source shortest path problem (SSSP) on an $n \times n$ grid graph with diagonals, see Fig. 8. Each node has a directed edge of length 1 to its right and to its above neighbor and an edge of length $\sqrt{2}$ to its upper right diagonal neighbor. The source node is the lower left node of the grid graph.

We used the LEDA implementation of Dijkstra's algorithm. In the considered problem, the algorithm must compare lengths of paths which involve sums of $\sqrt{2}$s. Since for most nodes there is more than one shortest path, the algorithm must detect equality between shortest path lengths. The running time depends heavily on the way lengths are assigned to the edges. We distinguished two cases. In the first case (called *different-constants case*), for each edge, a new instance of the number $\sqrt{2}$ is created. In the second case (called *one-constant case*), the edge lengths of the diagonals are identical copies of the same constant with value $\sqrt{2}$.

Fig. 9 shows the running times. In the different-constants case the running times explode and only problems with $n < 8$ can be solved. In the one-constant case the behavior of `leda_reals` is much better and instances with $n = 100$ can be solved in about 30 seconds. The running time with `doubles` for $n = 100$ is about 0.05 seconds.

We next explain these numbers. In the different-constants case the separation bound computation in `leda_reals` grossly overestimates

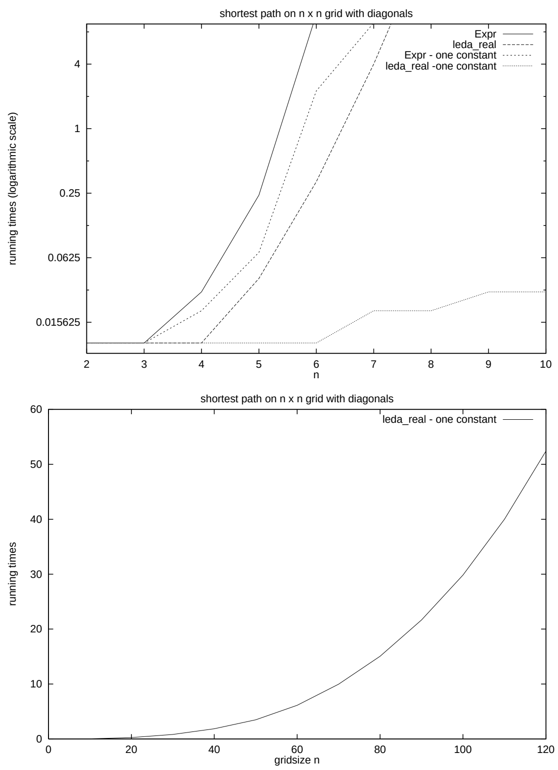

Figure 9: Running times for the shortest path problem on $n \times n$ grid graphs as a function of n. The upper figure uses a logarithmic scale for the running times and shows the running times times for the one-constant and the different-constants case for `leda_real` and `Real/Expr`. The lower figure shows the running time for `leda_real` for the one-constant case.

the degree of a sum like $\sqrt{2}+\sqrt{2}+\ldots+\sqrt{2}$. If the number of terms in the sum is N the degree is estimated to be 2^N. Therefore, the running time grows exponentially with n. It would be nice to extend the implementation of `leda_real`s such that several occurrences of the same radical are identified automatically. For example, one could keep a dictionary of all used radicals.

In the one-constant case the behavior is much better. `leda_real`s notice that 2 is (an upper bound on) the degree of the minimal polynomial of all path lengths, which is correct because all path lengths are in the quadratic extension field $\mathbb{Q}(\sqrt{2})$. Consider the comparison of the length of paths p_1 and p_2. The separation bound computed by `leda_real`s is the inverse of the sum of the lengths of the two paths; see Section 2. Therefore the separation bound grows linearly in n and hence the number of bits required for the computation of (approximations of) paths lengths grows logarithmically in n.

Let us take a closer look at the running time of the SSSP as a function of n. Note that the grid graph has size $\Theta(n^2)$. With floating-point arithmetic the running time of the Dijkstra algorithm using Fibonacci heaps is $O(n^2 \log n)$. With `leda_real`s the running time is $O(n^3 \log n \log\log n)$. This can be seen as follows: The shortest path computation builds an expression dag with one node for each arithmetic operation. Each node of this dag corresponds to the distance value of a vertex of the grid graph. Each comparison between two path lengths constructs a temporary node. The separation bound is computed by a walk over the expression dag. It is not sufficient to look at the children of the new nodes only, because one has to know which roots the dags of the children have in common[6]. The time for computing the separation bound is proportional to the size of the visited part of the dag, which is $O(n)$. Actually, the separation bound for a dag node, whose sign is not known yet, is recomputed whenever the precision is doubled. This is done because during recomputation with higher precision the value of a subexpression might be detected to be zero which would lead to a better separation bound. Since the maximum precision grows logarithmically in n, the precision is doubled $O(\log\log n)$ times during a sign computation. Since there are $O(n^2 \log n)$ distance comparisons, the running time for all separation bound computations is $O(n^3 \log n \log\log n)$.

In the shortest path computation many comparisons are between equal numbers and hence the precision is dictated by the separation bound. The computation of the value at a node may trigger the recomputation of the value at its children. Since each recomputation increases the precision of the value of the node by at least one bit and since the maximal precision required grows logarithmically in n we conclude that no node is recomputed more than $O(\log n)$ times. The recomputation of the value of a node from the values at the children requires a constant number of arithmetic operations. We conclude, that the overall time needed for approximating the values of all nodes in the dag is in $O(n^2 \log n \cdot M(\log n))$ where $M(k)$ is the time required to multiply two k-bit integers.

5 Conclusion

We have evaluated the performance of using the number type `leda_real` in the generic programming framework of CGAL. `leda_real`s offer a convenient and (reasonably) efficient way of comput-

[6] A way to avoid this walk over the dags is to store with each node the roots arising in the corresponding dag sorted according to some order. The roots for a new node can then be computed in time proportional to the sum of the number of roots of the children. The number of roots is usually small; if it is not, it takes too long to compute the sign of a zero expression anyway.

ing with expressions involving roots. They are superior to alternative packages. A challenge for the future is to improve the implementation of `leda_real`s such that the gap to the rational geometry kernel of LEDA becomes smaller. Such an improvement could be based on a faster computation of the initial approximation of a `leda_real`.

Acknowledgment: We thank Michael Seel for his advice during the experimental evaluation.

References

[1] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inform. Process. Lett.*, 7(5):219–222, 1978.

[2] J. Blömer. A probabilistic zero-test for expressions involving roots of rational numbers. In *Proc. of ESA '98*, pages 151–162, 1998.

[3] H. Brönniman, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. Research Report MPI-I-98-1-030, Max-Planck-Insitut für Informatik, 1998.

[4] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Companion page to 'Efficient exact geometric computation made easy'. `http:/www.mpi-sb.mpg.de/~stschirr/exact/made_easy`.

[5] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. *Proc. SODA97*, pages 702–709, 1997.

[6] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. In *Proc. SoCG98*, pages 175–183, 1998.

[7] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. SoCG95*, pages C18–C19, 1995.

[8] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class `real` number. Research Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, 1996. A more recent documentation of the implementation is available at `http://www.mpi-sb.mpg.de/~burnikel/reports/real.ps.gz`.

[9] CGAL project. `http://www.cs.uu.nl/CGAL/`.

[10] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224 – 242, 1971.

[11] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel : a basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 191–202. Springer LNCS 1148, 1996.

[12] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. Research Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, 1998.

[13] S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81 – 128. Information Geometers Ltd., 1993.

[14] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15:223–248, 1996.

[15] C. Hoffmann. The problem of accuracy and robustness in geometric computation. *IEEE Computer*, pages 31–41, March 1989.

[16] V. Karamcheti, C. Li, I. Pechtanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, 1999.

[17] M. Karasick, D. Lieber, and L. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

[18] J. Keyser. personal communication.

[19] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. `http://www.cs.unc.edu/~geom/MAPC`.

[20] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[21] K. Mehlhorn and S. Näher. *LEDA – A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, England, 1999.

[22] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User manual*, 3.7 edition, 1998. see `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[23] D. Michelucci. A quadratic non-standard arithmetic. In *Proc. 9th Canadian Conf. on Comp. Geom.*, 1997. `http://www.emse.fr/~micheluc/english/quadratique.html`.

[24] K. Ouchi. Real/Expr: Implementation of exact computation. Courant Institute, New York University, 1997. Master thesis. `http://cs.nyu.edu/exact/realexpr`.

[25] M. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering (WACG96)*, pages 53–58. Springer LNCS 1148, 1996.

[26] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *10th Symposium on Computer Arithmetic*, pages 132 – 143. IEEE Computer Society Press, 1991.

[27] A. Rege. *APU User Manual – Version 2.0*, 1996. `http://www.cs.berkeley.edu/~rege/apu/apu.html`.

[28] S. Schirra. Precision and robustness in geometric computations. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer LNCS 1340, 1997.

[29] S. Schirra. A case study on the cost of geometric computing. In *Proc. of ALENEX99*, 1999.

[30] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry* 18:305–363, 1997.

[31] R. C. Veltkamp. Generic programming in cgal, the computational geometry algorithms library. In *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997.

[32] C. K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7:3–23, 1997.

[33] C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook in Computational Geometry*, pages 653–668. CRC Press, 1997.

[34] C. K. Yap and T. Dubé. The exact computation paradigm. In D. Du and F. Hwang, editors, *Computing in Euclidean Geometry*, pages 452–492. World Scientific Press, 1995. 2nd edition.