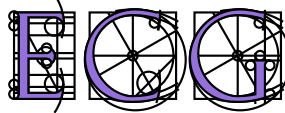


**ECG**  
*IST-2000-26473*  
Effective Computational Geometry for Curves and Surfaces



ECG Technical Report No. : *ECG-TR-363111-01*

*Effects of a Modular Filter on Geometric Applications*

M. Hemmer

L. Kettner

E. Schömer

Deliverable: 36 31 11 (item 01)  
Site: MPI  
Month: 36



Project funded by the European Community  
under the “Information Society Technologies”  
Programme (1998–2002)





# Effects of a Modular Filter on Geometric Applications\*

Michael Hemmer  
Elmar Schömer

Lutz Kettner

Johannes Gutenberg Universität  
Staudingerweg 9  
55099 Mainz, Germany

Max-Planck-Institut für Informatik  
Stuhlsatzenhausweg 85  
66123 Saarbrücken, Germany

## Abstract

We report on the effects of a filter based on modular arithmetic that has been introduced recently into the EXACUS library. Our experiments with planar arrangements for curves up to degree four show that the exact construction and comparison of real algebraic numbers are some of the most time consuming operations when solving intersection problems for curved objects. In our experiments the modular filter accelerated the computation for arrangements of cubic curves by a factor of about 6.

## 1 Introduction

The current way to speed up exact algorithms is to use rounded evaluation with a certified error to answer safely and quickly the easy cases, which already gave good results in practice [5, 10, 15, 8]. For non-degenerate situations these methods can reach almost the same running times as conventional floating point arithmetic does. But for degenerate and nearly degenerate situations verified floating point arithmetic is not applicable. Thus in general we have to switch to some costly exact arithmetic to decide these cases.

In our case, computing arrangements of algebraic curves [3, 9, 2], we have to insure that two univariate polynomials  $f$  and  $g$  do not have a nontrivial common factor or that a univariate polynomial  $f$  is squarefree. Otherwise we are forced to do a costly  $gcd$  computation, which is currently the main bottleneck in our approach.

A polynomial  $f$  is squarefree, if  $f$  and its derivative  $f'$  have a nontrivial common factor. Thus this question also reduces to whether two polynomials have a nontrivial common factor. Two polynomials  $f, g$  in  $\mathbb{Z}[x]$  have a common factor iff the *resultant* of  $f$  and  $g$ ,  $res(f, g) \in \mathbb{Z}$  is zero. The resultant can be e.g. computed as the determinant of the Bezoutian matrix. Of course this is not as expensive as a total  $gcd$  computation, but as our experiments show, the time spent in this step is still significant.

A first idea is to evaluate the determinant using interval arithmetic, but we expect this method to be susceptible to ill-conditioned matrices. The key observation is, that we only need to know whether the resultant is *not* zero. Therefore in most cases it is enough to evaluate the determinant in  $\mathbb{Z}/p\mathbb{Z}$ , for some prime  $p$ , to see that the resultant is not zero. The modular arithmetic is implemented using double arithmetic, based on techniques described in [6]. This approach is only 3 times slower than double arithmetic. Thus we can expect it to be even faster than interval arithmetic. For more details on the modular filter see section 3.

---

\*Partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces)

The test environment for our experiments on the modular filter was the EXACUS package CUBIX , for details on this approach see [9, 4]. We give a short overview on that in section 2 and outline the critical steps, in which the filter comes into account. Section 3 describes the different filters. The report on the experiments can be found in section 4.

## 2 The sweep-line algorithm for cubic curves

In this section we give a very short overview on the approach described in [9], but will especially outline the parts in which the filter comes into account.

An *algebraic curve* is defined in the following way: Let  $f$  be a polynomial in  $\mathbb{Q}[x, y]$ . We set  $\text{ZERO}(f) := \{(\alpha, \beta) \in \mathbb{R}^2 \mid f(\alpha, \beta) = 0\}$  and call  $\text{ZERO}(f)$  the *algebraic curve* defined by  $f$ . A *cubic curve*, or *cubic* for short, is an algebraic curve of degree 3. If the context is unambiguous, we will often identify the defining polynomial of an algebraic curve with its zero set.

The arrangement of a set  $F$  of cubic curves is computed using the Bentley-Ottmann sweep-line algorithm [1]. At each time during the sweep the curves intersect the sweep-line in a given order. While moving the sweep-line along the  $x$ -axis a change in the topology of the arrangement takes place iff this ordering changes. This happens only at a finite number of *event points*. The algorithm requires to insert the  $x$ -coordinates of event points into the  $X$ -structure, see [1]. These  $x$ -coordinates are the real roots of a univariate polynomial in  $x$ . This polynomial is the resultant of either two cubic curves or of one curve and its derivative. In case of cubics the degree of the resultant is bounded by 9.

For the predicates used by this approach it is very essential to know the multiplicity of the roots of the resultant. Thus if the resultant is not *squarefree* a factorization by multiplicities is required. This causes at least one *gcd* computation of the resultant and its derivative. Since the *gcd* computation is very time consuming, it is essential to precheck whether the resultant is not squarefree in order to avoid unnecessary *gcd* computations. This is the first place in which the modular filter described in section 3 comes into account. The other is within the comparison of algebraic numbers, which is described in the next section. For more details on the algorithm itself and the implementation of the predicates see [9].

### 2.1 Algebraic Numbers

Algebraic numbers are used to represent the  $x$ -coordinates of event points during the sweep, but are also used within the predicates of the algorithm. By definition, every real root of a univariate polynomial is a real *algebraic number*. In particular, a rational number is an algebraic number.

#### 2.1.1 Representation of algebraic numbers

We represent an algebraic number  $x$  as a triple  $(P, l, r)$  where  $P$  is a *squarefree* univariate polynomial with integer coefficients,  $l$  and  $r$  are rational numbers, such that  $P$  has exactly one real root in the open interval  $(l, r)$ , and  $P(l) \neq 0 \neq P(r)$ . Such an interval is called an *isolating interval* for the root. In case we know  $x$  to be rational, we store it as the degenerate interval  $(x, x)$ .

We determine isolating intervals by means of Uspensky's algorithm [7, 16], which is known to be the best method for squarefree polynomials [7].

#### 2.1.2 Refine isolating intervals

As  $x$  is a simple root of  $P$ , we can easily *refine* an isolating interval  $I = (l, r)$  at a given point  $m \in \mathbb{Q} \cap I$ , by evaluating the sign of  $P(m)$ . If  $P(m) = 0$ , we set  $I = (m, m)$ . Otherwise we set  $I = (l, m)$  if  $\text{sign}(P(l)) \neq$

$\text{sign}(P(m))$  or  $I = (m, r)$  if  $\text{sign}(P(l)) = \text{sign}(P(m))$ . Usually we refine the isolating interval by bisecting at the midpoint  $m = (l + r)/2$ .

### 2.1.3 Comparison of two algebraic numbers

We next describe how to compare two algebraic numbers  $x = (P, l_x, r_x)$  and  $y = (Q, l_y, r_y)$ . If the isolating intervals are disjoint, we just compare the intervals. Otherwise, let  $I = (l, r)$  be the intersection of the isolating intervals. We have  $x = y$  iff  $P$  and  $Q$  have a common root in  $I$ . We first refine the isolating intervals of  $x$  and  $y$  using the endpoints of  $I$ . Then it is either the case that both intervals are  $I$  or the intervals are disjoint. If they are disjoint, then we are done. Otherwise, we know that  $P$  and  $Q$  both have exactly one simple root in  $I$ . These roots are equal if  $G = \text{gcd}(P, Q)$  has a root  $z$  in  $I$ . We now use the fact that  $G$  has only simple roots. So  $x = y = z$  iff  $\text{sign}(G(l)) \neq \text{sign}(G(r))$ . Otherwise we can refine both isolating intervals for  $x$  and  $y$  as described in the preceding paragraph until they are disjoint.

### 2.1.4 Optimizations

- **Common factor propagation:** If  $G$  is nontrivial, we can use  $G$  to reduce the degree of the defining polynomials  $P$  and  $Q$ . If  $x = y$  we replace the defining polynomials by  $G$ , otherwise we replace them by  $P/G$  and  $Q/G$  respectively. Moreover we keep a list of all algebraic numbers defined by the same polynomial, thus we are able to propagate a discovered factor to all of them. This prevents us from computing the same  $\text{gcd}$  over and over again and in addition refines the representation of the involved algebraic numbers.

- **one-root-numbers**

We call an algebraic number a *one-root-number* if it is of the form  $\alpha + \beta\sqrt{\gamma}$  with  $\alpha, \beta, \gamma \in \mathbb{Q}$ . One-root-numbers can be represented explicitly using e.g. leda-real [14] or exchangeable CORE::Expr [12]. Integers are leda-reals, and if  $x$  and  $y$  are leda-reals, so are  $x \pm y$ ,  $x * y$ ,  $x/y$ , and  $\sqrt[k]{x}$  for arbitrary integer  $k$ . leda-reals have exact comparison operators  $\leq$ ,  $<$  and  $=$ . In particular, if  $x$  is a leda-real and  $P$  is a polynomial with integer coefficients, we can determine the sign of  $P(x)$ .

This enables us to compare two one-root-numbers  $x$  and  $y$  very efficiently just by using the comparison operator of the leda reals. If  $x = (P, l, r)$  and  $y$  is a leda-real, we proceed as follows: If  $y \leq l$  or  $y \geq r$ , the outcome of the comparison is clear. So assume  $l < y < r$ . If  $P(y) = 0$ , then  $x = y$ . Otherwise we can refine the isolating interval for  $x$  as described in the preceding paragraph until  $y \notin (l, r)$ . We notice that  $x$  is a *one-root-number* as soon as the degree of the defining polynomial  $P$  is  $\leq 2$ .

- **Prerefinement:**

New algebraic numbers often have quite large intervals, because Uspensky's algorithm stops as soon as root isolation is guaranteed. Therefore we extended the comparison by up to 16 prerefinement steps, before we conjecture an equality of two algebraic numbers. This especially compensates for the fact that we do not refine the algebraic numbers up to a certain precision in advance and in addition has the advantage that it is an adaptive approach.

- **Modular filter:** In most cases of inequality  $P$  and  $Q$  do not have a common factor at all. In these cases a computation of the GCD is time consuming and absolutely useless. It is enough to know that  $P$  and  $Q$  do not have a common factor. Thus we can use the filter described in section 3 to decide quickly on that fact. While the prerefinement succeeds in non-degenerate situations, it turned out, that especially for nearly degenerate situations applying the filter still gives a significant speed up as you'll see in section 4.

### 3 Modular Filter

The goal of the filter is to indicate quickly, that two polynomials  $f, g \in \mathbb{Z}[x]$  do not have a nontrivial common factor.

The modular filter is based on the following

**Theorem 1.** *Let  $f, g$  be two polynomials in  $\mathbb{Z}[x]$  and let  $\Phi: \mathbb{Z} \rightarrow \mathbb{Z}/p\mathbb{Z}$  be the canonical homomorphism from  $\mathbb{Z}$  to  $\mathbb{Z}/p\mathbb{Z}$  for some prime  $p$ . Then  $f$  and  $g$  do not have a common factor if  $\text{res}(\Phi(f), \Phi(g)) \neq 0 \pmod{p}$ .*

*Proof.* We use the well known fact that two polynomials  $f, g \in \mathbb{Z}[x]$  have a nontrivial common factor iff the resultant  $\text{res}(f, g) \in \mathbb{Z}$  is zero. And since  $\Phi$  is a homomorphism we know that

$$\begin{aligned} \text{res}(f, g) &= 0 \\ \Rightarrow \Phi(\text{res}(f, g)) &= 0 \pmod{p} \\ \Leftrightarrow \text{res}(\Phi(f), \Phi(g)) &= 0 \pmod{p} \end{aligned}$$

□

The modular filter applies  $\Phi$  to all entries of the Bezoutian matrix [13], and then computes the determinant in  $\mathbb{Z}/p\mathbb{Z}$ . If the outcome is not zero we can conclude that the polynomials  $f$  and  $g$  do not have a common factor.

The modular arithmetic is implemented using double arithmetic. It uses a fixed predefined prime  $p$  that fits in nearly half of the mantissa length for doubles. This allows us for example, to perform a multiplication of two such numbers at basically the cost of three double multiplications. For more details see [6].

The modular filter has two advantages. Firstly, it is only about three times slower than double arithmetic and therefore should be faster than interval arithmetic. Secondly, it has a very small probability to fail. Precisely, the probability to fail is equal to the probability that the result of the exact resultant is  $\in p\mathbb{Z}$ , which is very small since  $p > 2^{26}$ . In fact it never failed on any of our generated inputs as described in section 4.

Note that the probability that the modular filter fails is independent from the fact that the situation is degenerate or not. Thus it is especially applicable in nearly degenerate situations.

### 4 Experiments

We offer two series of benchmarks. Firstly, a random series to illustrate the general impact of the modular filter. Secondly, a series of nearly degenerate instances, to measure the impact of the modular filter on the comparison of algebraic numbers.

All times were taken on Intel Pentium 4 Mobile CPU 2.00GHz with 512 KB of cache running Linux. The compiler used was `g++-3.3.3 -O2`. The library used for exact computations was LEDA -4.4.1. We abdicated using also CORE for additional benchmarks, since the observed effects of the filter should be rather independent of the chosen library.

For each input size  $n$ , we have generated an odd number of candidate input data sets and picked the one with median average running time for inclusion in our benchmark.

## 4.1 Random series

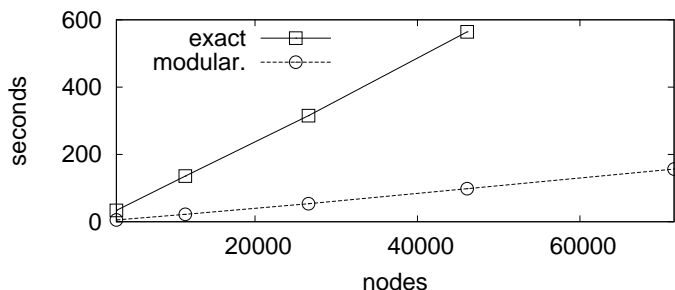
For each instance we generated  $n$  *random* cubic curves. Each curve  $f$  is defined by interpolation through 9 points chosen uniformly at random from a set of  $9n$  random points on the  $\{-128, \dots, 127\}^2$  integer grid. Every interpolation point results in a homogeneous linear condition on the 10 unknown coefficients of  $f$ , so that generically 9 conditions determine the equation of a curve uniquely, up to a constant factor. For this series the average coefficient bit size was 99.

To compare our results, we also implemented the `exact` version of the filter. It is implemented in the same way as the modular filter, besides that the determinant of the Bezoutian matrix is evaluated using exact arithmetic.

The table below reports on the overall running time of the sweep line algorithm for the random instances. The row labeled with `n` reports on the number of input curves, `nodes` states the number of nodes in the computed arrangement to reflect the output size of the algorithm. The rows `exact` and `modular` report on the running time of the sweep for the respective filter. The running times are given in seconds.

The plot below shows the running times as a function of the number of computed nodes (output complexity). In accordance with the theoretical analysis, see [9], the output complexity looks almost linear. However, the output size is quadratic in the number of curves, as for the straight-line case.

series	$n$	nodes	exact	modular
random	30	2933	33.62	5.41
random	60	11417	135.8	21.91
random	90	26579	315.13	53.5
random	120	46117	564.25	98.36
random	150	71594	-	156.62



For this series of random instances the modular filter has accelerated the computation by a factor of about 6.

A first look on the time spent within the comparison of two algebraic numbers showed, that for this series the modular filter had no effect. But for the random examples this is not surprising, since the prerefinement in the comparison of algebraic numbers succeeds in the most cases.

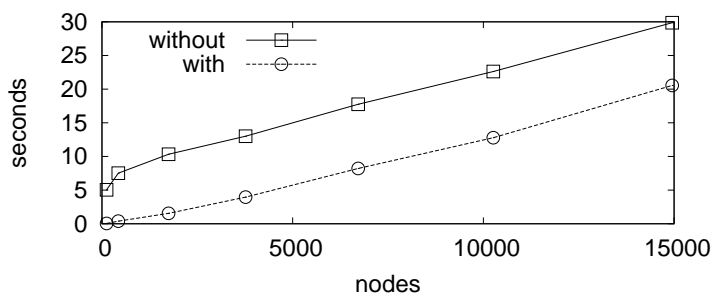
## 4.2 Nearly degenerate series

To expose the effect of the modular filter within the algebraic numbers we generated a series of *nearly degenerate* instances. It was obtained in a similar fashion as the random series, but only up to 16 interpolation points were used. Then the interpolation points of each curve were perturbed slightly. This resulted in nearly degenerate arrangements, with very dense clusters of intersection points around the original interpolation points. For this series the average coefficient bit size was 570.

For this series we always used the modular filter within the sweep, since it proved its profitability in the previous series. The first run used the algebraic numbers `without` the modular filter, while the second run used the algebraic numbers `with` the modular filter.

The table below reports on the running time spend within the comparison of algebraic numbers for the nearly degenerate instances. The running times are given in seconds.

series	$n$	nodes	without	with
n_degen	05	120	5.05	0.05
n_degen	10	429	7.52	0.38
n_degen	20	1746	10.33	1.55
n_degen	30	3765	13.76	3.96
n_degen	40	6718	17.76	8.21
n_degen	50	10259	22.62	12.79
n_degen	60	14950	29.86	20.55



The plot for the run `without` the modular filter shows a constant offset compared to the plot for the benchmarks `with` the modular filter. This is obviously caused by unnecessary *gcd* computations the modular filter was able to avoid. But why is it only a constant offset? The answer is, that the first algebraic numbers inserted in the X-structure have large intervals. Therefore the prerefinement fails, and since the modular filter is switched off, an unnecessary *gcd* is computed. But after a while several algebraic numbers are already inserted into the X-structure. Therefore the algebraic number that are inserted later are refined by the already very small intervals of the old algebraic numbers and therefore the prerefinement succeeds again. This is obviously a result of the way we generated the input since for every instance only 16 clusters of nearly degenerate intersections exist.

This example also shows the advantages of the adaptive approach we chose for the algebraic numbers, since they can somehow "react" on the peculiarities of an arrangement. But we must be aware of the fact, that prerefinement of isolating intervals can also have adverse effects: When equal but non-identical algebraic numbers are compared too often, it is conceivable that their repeated useless refinement and the increased cost of later operations caused by the increased bit size of the interval boundaries outweighs the gain of prerefinement.

## 5 Conclusions and further work

The modular filter proved to be very fast and at the same time very strict, since its low probability to fail is even independent from the fact that the situation is degenerate or not. Currently it is an indispensable filter to check squarefreeness of polynomials.

In the case of algebraic numbers prerefinement proved to be an very good filter that succeeds in most cases. But since it also has negative effects in case of equal algebraic numbers the modular filter is an optimal addendum that allows us to keep the number of prerefinements small enough to be tolerable.

The optimal number of prerefinements obviously depends on the structure of each arrangement. Up to now this somehow "well chosen" number of 16 behaved well in several instances but should be the matter of further investigations.

Another possibility to deal with algebraic numbers is to use the *diamond operator* from the extended *leda-reals* [11]. In fact we are currently in the process of testing this number type as another concept of algebraic numbers. But first tests indicate that this approach might be slower than the algebraic numbers described here.

## References

- [1] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.

- [2] Eric Berberich. *Exact Arrangements of Quadric Intersection Curves*. Universität des Saarlandes, Saarbrücken, 2004. Master Thesis.
- [3] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Kurt Mehlhorn, and Elmar Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *ESA 2002, Lecture Notes in Computer Science*, pages 174–186, 2002.
- [4] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, Dennis Weber, and Nicola Wolpert. Exacus: Efficient and exact algorithms for curves and surfaces. Technical Report ECG-TR-361200-02, MPI Saarbrücken, 2004. [www.mpi-sb.mpg.de/projects/EXACUS/](http://www.mpi-sb.mpg.de/projects/EXACUS/).
- [5] Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion. Interval arithmetic yields efficient arithmetic filters for computational geometry. In *Abstracts 14th European Workshop Comput. Geom.*, pages 49–54. Universitat Polyècnica de Catalunya, Barcelona, 1998.
- [6] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.
- [7] G. E. Collins and A.-G. Akritas. Polynomial real root isolation using Descartes’ rule of sign. In *SYMSAC*, pages 272–275, Portland, OR, 1976.
- [8] Olivier Devillers and Sylvain Pion. Efficient exact geometric predicates for Delaunay triangulations. Research Report ECG-TR-123102-01, INRIA Sophia-Antipolis, 2002.
- [9] Arno Eigenwillig, Lutz Kettner, Elmar Schömer, and Nicola Wolpert. Complete, exact, and efficient computations with cubic curves. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, 2004.
- [10] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.
- [11] Stefan Funke, Kurt Mehlhorn, and Susanne Schmitt. The leda class real number – extended version. Technical Report ECG-TR-363110-01, MPI Saarbrücken, 2004.
- [12] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee Yap. *The CORE Library Project*, 1.2 edition, 1999. <http://www.cs.nyu.edu/exact/core/>.
- [13] S. Y. Ku and R. J. Adler. Computing polynomial resultants: Bezout’s determinant vs. collins’ reduced p.r.s. algorithm. *Commun. ACM*, 12(1):23–30, 1969.
- [14] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.
- [15] Sylvain Pion. Interval arithmetic: An efficient implementation and an application to computational geometry. In *Workshop on Applications of Interval Analysis to systems and Control*, pages 99–110, 1999.
- [16] F. Rouillier and P. Zimmermann. Efficient isolation of polynomial real roots. Technical Report 4113, INRIA, 2001.