Univariate Algebraic Kernel and Application to Arrangements

Sylvain Lazard[1], Luis Peñaranda[1], and Elias Tsigaridas[2]

[1] INRIA Nancy - Grand Est, LORIA, France.
[2] INRIA Sophia-Antipolis - Méditerranée, France.
`FirstName.LastName@inria.fr`

Abstract. We present a CGAL-based univariate algebraic kernel, which provides certified real-root isolation of univariate polynomials with integer coefficients and standard functionalities such as basic arithmetic operations, greatest common divisor (gcd) and square-free factorization, as well as comparison and sign evaluations of real algebraic numbers.
We compare our kernel with other comparable kernels, demonstrating the efficiency of our approach. Our experiments are performed on large data sets including polynomials of high degree (up to 2 000) and with very large coefficients (up to 25 000 bits per coefficient).
We also address the problem of computing arrangements of x-monotone polynomial curves. We apply our kernel to this problem and demonstrate its efficiency compared to previous solutions available in CGAL.

1 Introduction

Implementing geometric algorithms robustly is known to be a difficult task for two main reasons. First, all degenerate situations have to be handled and second, algorithms often assume a real-RAM model (a random-access machine where each register can hold a real number and each arithmetic operation has unit cost) which is not realistic in practice. In recent years, the paradigm of exact geometric computing has arisen as a standard for robust implementations [24]. In this paradigm, geometric queries, also called predicates, such as "is a point inside, outside or on a circle?", are made exactly using, usually, either (i) exact arithmetic combined, for efficiency, with interval arithmetic on doubles or (ii) interval arithmetic on arbitrary-fixed-precision floating-point numbers combined with separation bounds; on the other hand, geometric constructions, such as the circle through three points or points of intersection between two curves, may be approximated.

We address here one recurrent difficulty arising when implementing algorithms dealing, in particular, with curved objects. Such algorithms usually require evaluating, manipulating and solving systems of polynomials equations and comparing their roots. One of the most critical parts of dealing with polynomials or polynomial systems is the isolation of the real roots and their comparison.

We restrict here our attention to the case of univariate polynomials and address this problem in the context of CGAL, a C++ Computational Geometry

Algorithms Library, which is an open source project and became a standard for the implementation of geometric algorithms [4].

CGAL is designed in a modular fashion following the *paradigm of generic programming*. Algorithms are typically parameterized by a *traits* class which encapsulates the geometric objects, predicates and constructions used by the algorithm. Algorithms can thus typically be implemented independently of the type of input objects. For instance, the core of a line-sweep algorithm for computing arrangements of plane curves [7] can be implemented independently of whether the curves are lines, line segments, or general curves; on the other hand, the elementary operations that depend on the type of the objects (such as, comparing x-coordinates of points of intersection) are implemented separately in traits classes. Similarly, the model of computation, such as exact arbitrary-length integer arithmetic or approximate fixed-precision floating-point arithmetic, are encapsulated in the concept of *kernel*. An implementation is thus typically separated in three or four layers, (i) the geometric algorithm which relies on (ii) a traits class, which itself relies on (iii) a kernel for elementary (typically geometric) operations. CGAL provides several predefined Cartesian kernels, for instance allowing standard Cartesian geometric operations on inputs defined with doubles and providing approximate constructions (*i.e.*, defined with double) but exact predicates. However, a kernel can also rely on (iv) a number type which essentially encapsulates the type of number (such as, double, arbitrary-length integers, intervals) and the associated arithmetic operations. A choice of traits classes, kernels and number types is useful as it gives freedom to the users and it makes it easier to compare and improve the various building blocks of an implementation.

Our Contributions. We present in this paper a CGAL-compliant algebraic kernel that provides real-root isolation of univariate integer polynomials and basic operations, *i.e.* comparisons and sign evaluations, of real algebraic numbers. This open-source kernel follows the CGAL specifications for algebraic kernels [3]. The root isolation is based on the interval Descartes algorithm [5] and uses the library RS [19]. Moreover, our kernel provides various operations for polynomials, such as gcd, which are crucial for manipulating algebraic numbers.

We compare our kernel with other comparable kernels and demonstrate the efficiency of our approach. We perform experiments on large data sets including polynomials of high degree (up to 2 000) and with very large coefficients (up to 25 000 bits per coefficient).

Finally, we apply our kernel to the problem of computing arrangements of x-monotone polynomial curves and demonstrate its efficiency compared to previous solutions available in CGAL.

Related work. Combining algebra and geometry for manipulating non-linear objects has been a long-standing challenge. Previous work includes, but it is not limited to, MAPC [14] a library for manipulating points that are defined algebraically and handling curves in the plane. More recently, the library EXACUS [2], which handles curves and surfaces in computational geometry and supports various algebraic operations, was developed and partially integrated into CGAL.

The notion of algebraic kernel for CGAL was proposed in 2004 [11]; in this work, the underlying algebraic operations were based on the SYNAPS library [15]. Several methods and algebraic kernels have been developed since then.

One kernel was developed by Hemmer and Limbach [13] following the generic programming paradigm using the C++ template mechanism. This kernel is templated by the representation of algebraic numbers and by the real root isolation method, for which two classes have been developed; one is based on the Descartes method and the other on the Bitstream Descartes method [9]. This approach has the advantage to allow, in principle, using the best instances for both template arguments.

Another kernel developed at INRIA relies on the SYNAPS library [15]. In this kernel there are several approaches concerning real root isolation, *i.e.*, methods based on Sturm subdivisions, sleeves approximations, continued fractions, and a symbolic-numeric combination of the sleeve and continued fractions methods (see [10]). Moreover, there are specialized methods for polynomials of degree less or equal than four [21].

★

⟦Luis says: Elias proposed the following for the next version: SYNAPS is currently integrated as the package `realroot`, in `mathemagix`[3], which is an open source effort for algebraic computations that combines symbolic and numeric computations. ⟧

Emiris et al. [10] presented some benchmarks of these various approaches in these two kernels as well as some tests on the kernel we present here. The authors mention that our kernel based on interval Descartes performs similarly to one approach (refer to as NCF2) based on continued fractions [20] for coefficients with (very) large bitsize but NCF2 is more efficient for small bitsize. They conclude that, first, dedicated algorithms for polynomials of degree less than (or equal to) four is always the most efficient approach and, second, that NCF2 always perform the best except for low-degree and high-bitsize polynomials, in which case the kernel based on the Bitstream Descartes method performs the best. We moderate here these conclusions.

The rest of the paper is structured as follows. In the next section we describe our algebraic kernel. In Section 3, we present various experiments on the isolation of real roots and on the comparison of algebraic numbers. In Section 4, we apply our kernel to the problem of computing arrangements and compare it to previous solutions available in CGAL. We finally conclude in Section 5.

2 Univariate algebraic kernel

We describe here our implementation of our univariate algebraic kernel. The two main requirements of the CGAL specifications, which we describe here, are the isolation of real roots and their comparison. We also describe our implementation of two operations, the gcd computation and the refinement of isolating intervals, that are both needed for comparing algebraic numbers.

[3] http://www-sop.inria.fr/galaad/mathemagix/

Preliminaries. The kernel handles univariate polynomials and algebraic numbers. The polynomials have integer coefficients and are represented by arrays of GMP arbitrary-length integers [12]. We implemented in the kernel the basic functions for polynomials. An algebraic number that is a root of a polynomial F is represented by F and an isolating interval, that is an interval containing this root but no other root of F. We implemented intervals using the MPFI library [16], which represents intervals with two MPFR arbitrary-fixed-precision floating-point numbers [17]; note that MPFR is developed on top of the GMP library for multi-precision arithmetic [12].

Root isolation. For isolating the real roots of univariate polynomials with integer coefficients, we developed an interface with the library RS [19]. This library is written in C and is based on Descartes' rule for isolating the real roots of univariate polynomials with integer coefficients.

We briefly detail here the general design of the RS library; see [18] for details. RS is based on an algorithm known as *interval Descartes* [5]; namely, the coefficients of the polynomials obtained by changes of variable, sending intervals $[a, b]$ onto $[0, +\infty]$, are only approximated using interval arithmetic when this is sufficient for determining their signs. Note that the order in which these transformations are performed in RS is important for memory consumption. The intervals and operations on them are handled by the MPFI library.

Algebraic number comparison. As mentioned above, one of the main requirements of the CGAL algebraic kernel specifications is to compare two algebraic numbers r_1 and r_2. If we are lucky, their isolating intervals do not overlap and the comparison is straightforward. This is, of course, not always the case. If we knew that they were not equal, we could refine both isolating intervals until they are disjoint. Hence, the problem reduces to determining whether the algebraic numbers are equal or not.

To do so, we compute the square-free factorization of the gcd of the polynomials associated to the algebraic numbers. The roots of this gcd are the common roots of both polynomials. We calculate the intersection, I, of the isolating intervals of r_1 and r_2. The gcd has a root in this interval if and only if $r_1 = r_2$.

To determine whether the gcd has a root in interval I, it suffices to check the sign of the gcd at the endpoints of I: if they are different or one of them is zero, the gcd has a root in I and $r_1 = r_2$; otherwise, $r_1 \neq r_2$ and we can refine both intervals until they are disjoint.

Gcd computations. Computing greatest common divisors between two polynomials is not a difficult task, however, it is not trivial to do so efficiently. A naive implementation of the Euclidean algorithm works fine for small polynomials but the intermediate coefficients suffer an exponential grow in size, which is not manageable for medium to large size polynomials. We thus implemented a *modular* gcd function. We did not use some existing implementations mainly for efficiency because converting polynomials from one representation to another is substantially costly as soon as the degree and bitsize are large. Our function calculates the gcd of polynomials modulo some prime numbers and reconstructs later the result with the help of the *Chinese remainder theorem.* (See *e.g.*, [23] for details.)

Refining isolating intervals. As we mentioned before, refining the interval representing an algebraic number is critical for comparing such numbers. We provide two approaches for refinement.

Both approaches require that the polynomial associated to the algebraic number is square free. The first step thus consists of computing the square-free part of the polynomial (by computing the gcd of the polynomial and its derivative).

Our first approach is a simple bisection algorithm. It consists in calculating the sign of the polynomial associated to the algebraic number at the endpoints and midpoint of the interval. Depending on these signs, we refine the isolating interval to its left of right half.

Our second approach is a quadratic interval refinement [1]. Roughly speaking, this method splits the interval in many parts and, based on a linear interpolation, guesses in which one the root lies. If the guess is correct, the algorithm divides in the next refinement step the interval in more parts and, if not, in less.

Unfortunately, even with our careful implementation this approach turns out to be, on average, only just a bit faster than the bisection approach. Our experiments showed that the bottleneck of the refinement is the evaluation of polynomials.

3 Kernel benchmarks

In this section, we analyze the running time of the two main functions of our algebraic kernel, that (i) isolate the roots of a polynomial and (ii) compare two algebraic numbers that is, compare the roots of two polynomials. We also compare the performance of our kernel with the one based on the Bistream Descartes method [9] and developed by Hemmer and Limbach [13] (referred to as MPII's kernel)[4] and with a kernel based on continued fractions [20] and developed on top of the SYNAPS library [15] (referred to as SYNAPS' kernel).

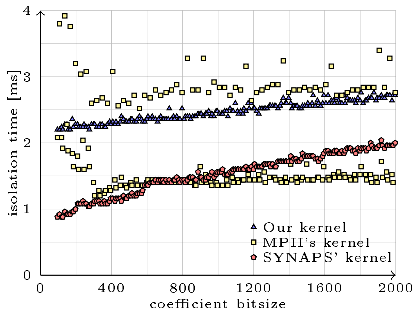
All tests were ran on a single-core 3.2 GHz Intel Pentium 4 with 2 Gb of RAM and 2048 kb of cache memory, using 64-bit Linux.

Root isolation. We consider two suites of experiments in which we either fix the degree of the polynomials and vary the bitsize of the coefficients or the converse; see Figs. 1 and 2. In each experiment, we report the running time for isolating all the roots per polynomial, averaged over different trials, for our kernel, MPII's and SYNAPS' kernel.

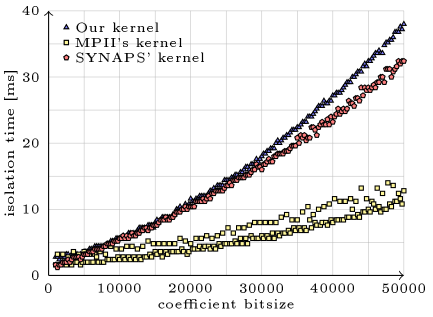
Varying bitsize. We study here polynomials with rather low degree (12) but with no complex root and polynomials with reasonably large degree (100) with random coefficients (and thus with few real roots).

The first test sets comes from [13]. See Fig. 1. It consists of polynomials of degree 12, each one being the product of six degree-two polynomials with

[4] We parameterized MPII's kernel to use Bitstream Descartes as root isolator, `algebraic_real_bfi_rep` as algebraic number representation and CORE integers and rationals to represent the coefficients of the polynomials and the isolation bounds of algebraic numbers, respectively. The choice of CORE (vs. LEDA) was induced by the need of testing the kernels in the same conditions, that is, relying on GMP.

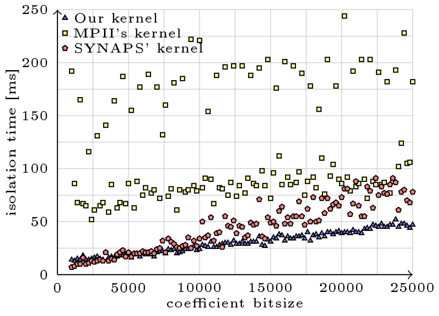


(a)

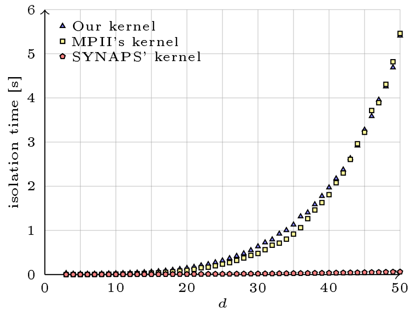


(b)

Fig. 1. Running time for isolating all the real roots of degree 12 polynomials with 12 real roots in terms of the maximum bitsize of their coefficients.



(a)



(b)

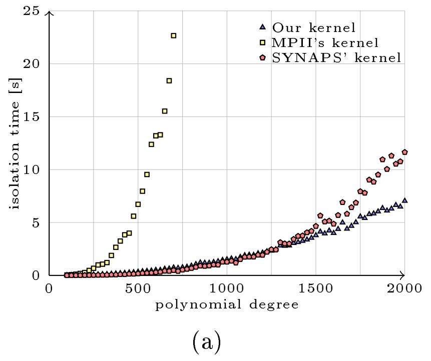
Fig. 2. Running time for isolating all the real roots of (a) degree 100 polynomials in terms of the maximum bitsize of their coefficients and (b) Mignotte polynomials of the form $f = x^d - 2(kx-1)^2$ in terms of the degree d.

two roots, at least one of them in the interval $[0,1]$; every polynomial thus has 12 real roots. We vary the maximum bitsize of all the coefficients of the input polynomial from 100 to 50 000 and average each test over 250 trials.

Secondly, we consider random polynomials with constant degree 100 and coefficients with varying bitsize. See Fig. 2(a). Note that such random polynomials have few roots: the expected number of real roots of a polynomial of degree d with coefficients independently chosen from the standard normal distribution is $\frac{2}{\pi}\ln(d) + C + \frac{2}{\pi d} + O(1/d^2)$ where $C \approx 0.625735$ [8]; this gives, for degree 100 an average of about 3.6 roots (note that this bound matches extremely well experimental observations). We vary the maximum bitsize of all the coefficients from 2 000 to 25 000 and average each test over 100 trials.

Varying degree. We consider two sets of experiments in which we study random polynomials and Mignotte polynomials (which have two very close roots).

We first consider polynomials with random coefficients of fixed bitsize for various values between 32 and 1 000. We then vary the degree of the polynomials from 100 to 2 000 and average our experiments over 100 trials (see Fig. 3). Note that the above formula gives an expected number of roots varying from 3.6 to



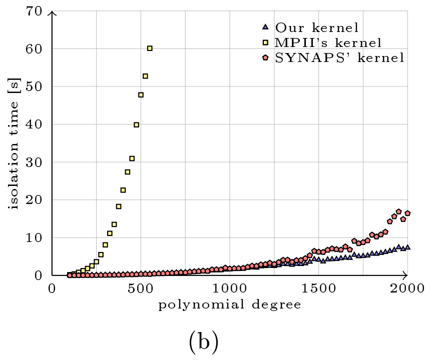


Fig. 3. Running time for isolating all the real roots of random polynomials with coefficients of bitsize (a) 32 and (b) 1000, and depending on the degree.

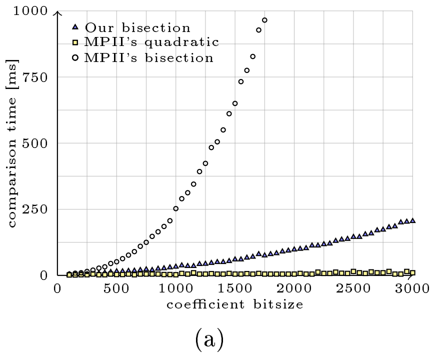
5.5. We observe that the running time is almost independent of the bitsize in the considered range.

Finally, we test Mignotte polynomials, that is polynomials of the form $x^d - 2(kx - 1)^2$. Such polynomials are known to be challenging for Descartes algorithms because two of their roots are very close to each other; the isolating intervals for these two roots are thus very small. For these tests, we used Mignotte polynomials with coefficients of bitsize 50, with varying degree d from 5 to 50. See Fig. 2(b). We averaged the running time over 5 trials for each degree. We observed essentially no difference between our kernel and MPII's one; they take roughly 0.2 and 5.5 seconds for Mignotte polynomials of degree 20 and 50, respectively. However, SYNAPS' kernel is much more efficient as the continued fractions algorithm is not so affected by the closeness of the roots.

Discussion. We observe (Fig. 1(a)) that SYNAPS' kernel is more efficient than both our and MPII's kernel in the case of polynomials of small degree (*e.g.*, twelve) and small to moderately large coefficients (up to 2 000 bits per coefficient). However, for extremely large coefficients MPII's kernel is substantially more efficient (by a factor of up to 3 for coefficients of up to 50 000 bits) than both our and SYNAPS' kernels, which perform similarly.

For polynomials of reasonable large degree, both our and SYNAPS' kernels are much more efficient that MPII's kernel; furthermore these two kernels behave similarly for degrees up to 1 500 and our kernel becomes more efficient for higher degrees (by a factor 2 for degree 2 000).

We also observe that the running time is *highly* dependent of the various settings. For instance, our kernel is up to 5 times slower when using approximate evaluation for high-degree and high-bitsize polynomials. Also, MPII's kernel is in some cases about 10 times slower when changing the arithmetic kernel to LEDA, the representation of algebraic numbers and some internal algorithms such as the refinement function. This explains why our benchmarks on both MPII's and SYNAPS' kernels are substantially better than in Emiris et al. experiments [10].



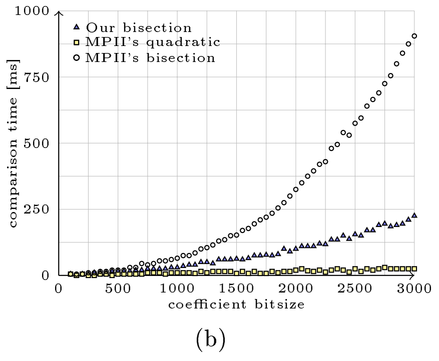


Fig. 4. Running time for comparing two distinct close roots of two almost identical polynomials of degree 20 with (a) no common roots and (b) a common factor of degree 10.

We also observe that the running time of MPII's kernel is unstable in our experiments (Figs. 1 and 2(a)); surprisingly, this instability occurs when the experiments are performed on a 64-bits architecture, but it is stable on 32-bits architecture as shown in previous experiments [10].

Comparison of algebraic numbers. We consider three suites of experiments for comparing algebraic numbers; see Fig. 4. Recall that an algebraic number ρ is here represented by a polynomial F that vanishes at ρ and an isolating interval containing ρ but no other root of F. Recall also that the comparison of two algebraic numbers is done by (i) testing whether the intervals are disjoint; if so, report the ordering, otherwise (ii) compute the gcd of the two polynomials and test whether the gcd vanishes in the intersection of the two intervals; if so, report the equality of the numbers, otherwise (iii) refine the intervals until they are disjoint.

First, we analyze the cost of trivial comparisons that is, when the two intervals are disjoint. For that we compare the roots of two random polynomials. We observe that, as expected, the comparison time is negligible and independent of both the degree of the polynomials and the bitsize of their coefficients.

Second, we analyze the cost of comparing roots that are very close to each other but whose associate polynomials have no common root. This case is expensive because we need to refine the intervals until they do not overlap; this is, however, not the worst situation because the gcd of the two polynomials is 1 which is tested efficiently with a modular gcd. We perform these experiments as follows. We generate pairs of polynomials, one with random coefficients and the other by only adding 1 to one of the coefficients of the first polynomial. Such polynomials are such that the i-th roots of both polynomials are very close to each other. We generate such pairs of polynomials with constant degree (equal to 20) and vary the maximum bitsize of the coefficients. As the bitsize increases, the pairs of roots that are close become even closer and thus the comparison time increases. The results of these experiments are presented in Fig. 4(a), which reports the average running time for comparing two close roots. We show in this figure three curves, one corresponding to our bisection algorithm, and two corresponding the two refinement methods implemented in the MPII's kernel: the usual bisection and a quadratic refinement algorithm.

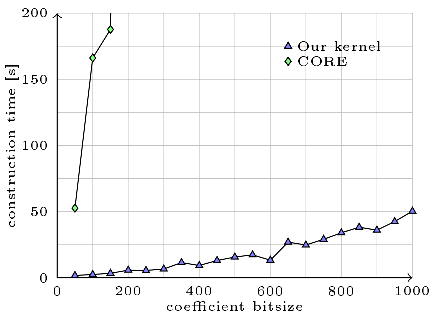
Third, we consider the, a priori, most expensive scenario in which we compare roots that are either equal or very close to each others and such that their associate polynomials have some roots in common. In this case, we accumulate the cost of computing a non-trivial gcd of the two polynomials with the cost of refining intervals when comparing two non-equal roots. In practice, we generate pairs of degree-20 polynomials each defined as the product of two degree-10 terms; one of these factors is random and common to the two polynomials; the other factor is random in one of the polynomials and slightly modified in the other polynomial where, slightly modified means, as above, that we add 1 to one of the coefficients. We then vary the maximum bitsize of the coefficients.

Discussion. We see in Fig. 4 that the MPII's quadratic refinement algorithm largely outperforms the two bisection methods. However, our bisection method is faster than MPII's one, by a factor up to 10. We also observed that the running time for comparing equal roots is negligible compared to the cost of comparing close but distinct roots. (The running time reported in Fig. 4(b) is actually the total time for comparing all pairs of roots divided by the number of comparisons of close but distinct roots.) This explains why our kernel behaves similarly in Figs. 4(a) and 4(b). Overall, it appears that comparing algebraic numbers that are very close is fairly time consuming and that the most time-consuming part of the comparison is the evaluation of polynomials performed during the interval refinements.

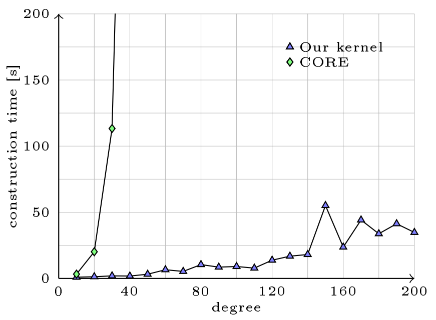
4 Arrangements

As an example of possible benefit of having efficient algebraic kernels in CGAL, we used our implementation to construct arrangements of polynomial functions. Wein and Fogel [22] provided a CGAL package for calculating arrangements of general curves which requires as parameter a *traits class* containing the data structures to store the curves and various primitive operations, such as comparing the relative positions of points of intersection. We implemented a traits class which uses the functions of our algebraic kernel and compared its performance with another traits classes which comes with CGAL's arrangement package and uses the CORE library [6].

In order to generate challenging data sets we proceed as follows. First we generate n random polynomials. To each of them we add 1 to the constant coefficient, m times, thus producing a data set of $n(m+1)$ univariate polynomials. Notice that the arrangement of the graphs of these polynomials is guaranteed to be degenerate, *i.e.*, there are intersections with the same x-coordinate. The arrangements generated this way have four parameters: the number n of initial polynomials, the number m of "shifts" that we perform, the degree d of the polynomials, and the bitsize τ of their coefficients. We ran experiments varying the values of the last three of these parameters and setting $n = 5$.



(a)



(b)

Fig. 5. Arrangements of five polynomials, shifted four times each, (a) of degree 20 and varying bitsize and (b) of bitsize 32 and varying degree.

Fig. 5(a) shows the running time in terms of the bitsize τ for a data set where $d = 20$ and $m = 4$ (giving 25 polynomials). Fig. 5(b) shows the running time in terms of the degree d for a second data set where $\tau = 32$ and $m = 4$. We see from these experiments that running time using CORE is considerably higher than when using our kernel. We also make the following observations.

Fig. 5(a) shows that the running time depends on the bitsize. When we change the bitsize of the coefficients of the random polynomials, the size of the arrangement does not change; that means that the number of comparisons and root isolations the kernel must perform is roughly the same in all the arrangements of the test suite. The isolation time for random polynomials does not depend much on the bitsize (as shown in Fig. 2(a)), but the comparison time does. It follows that the running time increases with the bitsize.

Fig. 5(b) shows that the running time depends also on the degree of the input polynomials. As we saw in Section 3, the expected number of real roots of a random polynomial depends on its degree. The size of the arrangement thus increases with the degree of the input polynomials: each vertex is the root of the difference between two input polynomials, therefore there will be more vertices. Thus, when we increment the degree of the inputs, the number of comparisons and isolations increases; furthermore, the running time for each of these operations increases with the degree of the input.

5 Conclusion

We presented a new CGAL-compliant algebraic kernel that provides certified real-root isolation of univariate polynomials with integer coefficients based on the interval Descartes algorithm. This kernel also provides the comparison of algebraic numbers and other standard functionalities.

We compared our kernel with other comparable kernels on large data sets including, for the first time, polynomials of high degree (up to 2 000) and with extremely large coefficients (up to 25 000 bits per coefficient). We demonstrated the efficiency of our approach and showed that it performs similarly, in most cases, with one kernel based on the SYNAPS library; more precisely, our kernel is more efficient for polynomials of very large degree (greater than 1 800) and less efficient for polynomials of very small degree and with small to moderate size coefficients. Also, our kernel is a lot more efficient that the kernel developed at MPII for polynomials of large degree (greater than 200); it is however less efficient for polynomials of small degree and with extremely large coefficients.

Our tests indicate that the kernel developed at MPII appears to be less efficient than the other two for polynomials of large degree. However it should be stressed that this kernel is the only one among the three that is templeted by the number type of the coefficients. Of course this does not imply that efficiency is necessarily lost by following the generic programming paradigm, but it does imply that, from the user point of view, some substantial gain of efficiency can sometimes be made by using a kernel that does not follow this paradigm.

We also compared the performance of the kernels on the comparison of algebraic numbers. We observed in these tests that the bisection algorithm runs much faster when it is specialized on a number type since it allows for low level optimizations, confirming thus the assertion in the previous paragraph. On the other hand, it becomes evident that the bisection method is not the most efficient algorithm when a large number of refinements is needed, and MPII's quadratic refinement is the fastest method by far.

A fairly large choice of algebraic kernels and, in particular, of methods for isolating the real roots of polynomials, is now available in CGAL. This allows, in particular, to compare and improve the various methods. It appears that between the two big classes of methods, based on continued fractions and Descartes algorithms, neither is clearly much better than the other. However, some substantial differences appear between the various implementations, but, of course, it is always very difficult to benchmark implementations. For instance, we observed here that the running times are highly dependent of the various settings and architectures.

Finally, we also address the problem of computing arrangements of x-monotone polynomial curves. We apply our kernel to this problem and demonstrate its efficiency compared to previous solutions available in CGAL.

Acknowledgments

The authors are grateful to F. Rouillier, Z. Zafeirakopoulos, M. Hemmer, E. Berberich, M. Kerber, and S. Limbach for fruitful discussions and suggestions.

References

1. J. Abbott. Quadratic interval refinement for real roots. In *International Symposium on Symbolic and Algebraic Computation (ISSAC), poster presentation*, 2006.
2. E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert. EXACUS: Efficient and Exact Algorithms for Curves and Surfaces. In *Proc. 13th Annual European Symposium on Algorithms (ESA)*, volume 1669 of *LNCS*, pages 155–166. Springer, 2005.
3. E. Berberich, M. Hemmer, M. Karavelas, and M. Teillaud. Revision of the interface specification of algebaic kernel. Technical Report ACS-TR-243301-01, ACS European Project, 2007.

4. Cgal, Computational Geometry Algorithms Library. http://www.cgal.org.
5. G. Collins, J. Johnson, and W. Krandick. Interval Arithmetic in Cylindrical Algebraic Decomposition. *Journal of Symbolic Computation*, 34(2):145–157, 2002.
6. The Core library. `http://cs.nyu.edu/exact/`.
7. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 3rd edition, 2008.
8. A. Edelman and E. Kostlan. How may zeros of a random polynomial are real? *Bulletin of American Mathematical Society*, 32(1):1–37, Jan 1995.
9. A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, and N. Wolpert. A Descartes Algorithm for Polynomials with Bit-Stream Coefficients. In *Proc. 8th Int. Workshop on Computer Algebra in Scient. Comput. (CASC)*, volume 3718 of *LNCS*, pages 138–149. Springer, 2005.
10. I. Emiris, M. Hemmer, M. Karavelas, S. Limbach, B. Mourrain, E. Tsigaridas, and Z. Zafeirakopoulos. Cross-benchmarks for univariate algebraic kernels. Technical Report ACS-TR-363602-02, ACS European Project, 2008.
11. I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards and open curved kernel. In *Proc. 20th Annual ACM Symp. on Computational Geometry (SoCG)*, pages 438–446, New York, USA, 2004.
12. Gmp, Gnu multiple precision arithmetic library. `http://gmplib.org/`.
13. M. Hemmer and S. Limbach. Benchmarks on a generic univariate algebraic kernel. Technical Report ACS-TR-243306-03, ACS European Project, 2006.
14. J. Keyser, T. Culver, D. Manocha, and S. Krishnan. Efficient and exact manipulation of algebraic points and curves. *Computer-Aided Design*, 32(11):649–662, 2000.
15. B. Mourrain, P. Pavone, P. Trébuchet, E. P. Tsigaridas, and J. Wintz. Synaps, a library for dedicated applications in symbolic numeric computations. In M. Stillman, N. Takayama, and J. Verschelde, editors, *IMA Volumes in Mathematics and its Applications*, pages 81–110. Springer, New York, 2007. `http://synaps.inria.fr`.
16. Mpfi, multiple precision interval arithmetic library. `http://perso.ens-lyon.fr/nathalie.revol/software.html`.
17. Mpfr, library for multiple-precision floating-point computations. `http://mpfr.org/`.
18. F. Rouillier and Z. Zimmermann. Efficient isolation of polynomial's real roots. *J. of Computational and Applied Mathematics*, 162(1):33–50, 2004.
19. Rs, a software for real solving of algebraic systems. F. Rouillier. `http://fgbrs.lip6.fr`.
20. E. P. Tsigaridas and I. Z. Emiris. On the complexity of real root isolation using Continued Fractions. *Theoretical Computer Science*, 392:158–173, 2008.
21. E. P. Tsigaridas and I. Z. Emiris. Real algebraic numbers and polynomial systems of small degree. *Theoretical Computer Science*, 409(2):186 – 199, 2008.
22. R. Wein and E. Fogel. The new design of CGAL's arrangement package. Technical report, Tel-Aviv University, 2005.
23. C. Yap. *Fundamental Problems of Algorithmic Algebra.* Oxford University Press, Oxford-New York, 2000.
24. C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41, pages 927–952. Chapmen & Hall/CRC, Boca Raton, FL, 2nd edition, 2004.