Towards an Open Curved Kernel *

Ioannis Z. Emiris †
Dept. Informatics & Telecoms
National University of Athens
Greece
emiris@di.uoa.gr

Athanasios Kakargias
Dept. Informatics & Telecoms
National University of Athens
Greece
grad0460@di.uoa.gr

Sylvain Pion ‡
INRIA
Sophia Antipolis
France
Sylvain.Pion@inria.fr

Monique Teillaud §
INRIA
Sophia Antipolis
France
Monique.Teillaud@inria.fr

Elias P. Tsigaridas ¶
Dept. Informatics & Telecoms
National University of Athens
Greece
et@di.uoa.gr

ABSTRACT

Our work goes towards answering the growing need for the robust and efficient manipulation of curved objects in numerous applications. The kernel of the CGAL library provides several functionalities which are, however, mostly restricted to linear objects.

We focus here on the arrangement of conic arcs in the plane. Our first contribution is the design, implementation and testing of a kernel for computing arrangements of circular arcs. A preliminary C++ implementation exists also for arbitrary conic curves. We discuss the representation and predicates of the geometric objects. Our implementation is targeted for inclusion in the CGAL library.

Our second contribution concerns exact and efficient algebraic algorithms for the case of conics. They treat all inputs, including degeneracies, and they are implemented as part of the library SYNAPS 2.1. Our tools include Sturm sequences, resultants, Descartes' rule, and isolating points.

Thirdly, our experiments on circular arcs show that our methods compare favorably to existing alternatives using CORE 1.6x and LEDA 4.5.

Categories and Subject Descriptors: D.2.13 [Reusable Software]: Reusable libraries; F.2.2 [Nonnumerical Algorithms and Problems]: Geometrical problems and computations; G.4 [Mathematical Software]: Reliability and robustness; I.1.2 [Algorithms]: Algebraic algorithms

General Terms: Algorithms, Design, Experimentation, Performance, Reliability

Keywords: Exact geometric computation, Arrangement, Conic arc, C++

1. INTRODUCTION

Curved objects are becoming increasingly important in computational geometry; one reason, is their wide range of applications, including those in solid modeling, CAD, molecular biology, GIS. Our work is inscribed in a general effort to extend current geometric software from linear to non-linear geometric objects. In particular, this paper describes our work in extending the CGAL [1] library with a kernel for curved objects and the related operations. It is clear that such a kernel relies heavily on real algebraic numbers, polynomial equations and systems of polynomials of bounded degree.

Our ultimate goal is to propose a modular and efficient approach to extend the CGAL library to manipulate curved objects. The CGAL *Kernel* provides the user mainly with *linear* objects: points, line segments, triangles... and basic operations on them. Circles and spheres are also defined but with very few functionalities. Curves are already present in the so called *traits classes* of certain specific packages of the CGAL *Basic library*: the arrangement package provides the user with a traits class for conic arcs, the optimization package comes with conics and basic operations on them needed for computing minimum enclosing ellipses, whereas the Apollonius graph package computes the Voronoi diagram of circles.

*Work 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), www-sop.inria.fr/prisme/ECG/. Also partially supported by INRIA's project "Calamata", a Team Association between the GALAAD group of INRIA and the Department of Informatics & Telecommunications, National University of Athens.

†http://www.di.uoa.gr/~emiris/index-eng.html

‡http://www-sop.inria.fr/geometrica/team/Sylvain.Pion/ Part of this work was conducted while the author was working at the Max Planck Institut für Informatik, Saarbrücken, Germany

§http://www-sop.inria.fr/galaad/teillaud/

¶http://theta.di.uoa.gr/~et/

SCG'04, June 8–11, 2004, Brooklyn, New York, USA.

[1] http://www.cgal.org

Our first contribution is the design and implementation of an open kernel for conics and related computations, that we aim at extending towards a more general curved kernel in the future. We show how it may represent the relevant geometric objects and how it can be interfaced with an algebraic module. The design and its implementation have striven for modularity and generality, and are thus parameterized by a geometric linear kernel (ie., for linear objects) and an algebraic kernel. Models (in the C++ Standard Template Library sense [1]) for these parameters are provided by CGAL and based on SYNAPS[2] respectively.

Existing work on arrangements of curves includes work related to the library EXACUS[3] [3, 6, 21] as well as work on the CGAL arrangement package [19]. Both of these compute arrangements of arbitrary conics, but only the second software is publicly available. Related work on surface arrangements includes [11, 16]. Another platform for curves and surfaces, with an emphasis towards geometric modeling, is being prepared [10].

Usually, the main issue is the implementation of the geometric primitives: predicates and constructions. They reduce to manipulations of roots of polynomials or polynomial systems. Existing libraries such as CORE[4] or LEDA[5] propose types that support *exact* comparisons of algebraic numbers of arbitrary degree, provided that they are specified by an explicit expression containing radicals or, more recently, as the root of a univariate polynomial (named the *rootOf* or *diamond* operator, respectively). These comparisons rely on lazy refinement of multi-precision approximations and *separation bounds* [4, 15]. The CGAL traits class for conic arcs of the arrangement package [19, 20] depends on such number types. Existing software by the computer algebra community (such as Maple or Axiom) offer generality but lack the efficiency required in our applications.

Alternative methods to achieve *exact* comparisons of algebraic numbers efficiently, by computing only *signs of polynomial expressions* in the input data, were studied in [5, 14] for second-degree polynomials and in [7, 8] for degree up to 4. These methods are based on general algebraic tools such as Sturm sequences, resultants and Descartes' rule. Our methods follow this approach and, in addition, use low-degree algebraic numbers for isolating the real roots of polynomials. In the case of conic curves, algebraic numbers are of degree at most 4, and the isolating points are always rational. It is important to note that the same methods should cover a wider class of geometric problems, ranging from Voronoi diagrams of curved objects [14] to computing with kinetic data structures [12]. A recent implementation of exact and general algebraic numbers is proposed in [12].

Our implementations are being made publicly available through CGAL and SYNAPS. We support our claims of efficiency by experimental evidence on circular arcs in 5 different configurations. We use the only publicly available implementation of arrangements, which is provided by CGAL, to compare our implementation with the CGAL traits class for conic arcs that uses LEDA::`real`. Our code is never slower and, on most configurations, much faster. We conclude that our methods yield efficient implementations and define a very promising approach for computing arrangements of conics.

The next section discusses the proposed kernel design, and Section 3 outlines the representation of the geometric objects and the geometric primitives. In Section 4 we discuss the required algebraic concepts, namely algebraic numbers and polynomials. This is made concrete in Section 5, where different models for the algebraic kernel are described. Our experimental data is presented in Section 6. We conclude with current and future work.

[2]http://www-sop.inria.fr/galaad/logiciels/synaps/
[3]http://www.mpi-sb.mpg.de/projects/EXACUS
[4]http://www.cs.nyu.edu/exact/core/
[5]http://www.algorithmic-solutions.com/enleda.htm

2. CURVED KERNEL DESIGN

We describe the C++ design chosen for organizing our code, and the kind of interface provided by our kernel. As in [17], our choices have been heavily inspired by the CGAL kernel design [13] which is extensible and adaptable. Indeed, one of its features is the ability to apply primitives like geometric predicates and constructions either to the geometric objects which are provided by our kernel, or to user-defined objects.

A primary concern is good interoperability with CGAL, and thus one of the goals was to be able to reuse the CGAL kernel for objects like points, circles and number types. Still, in the philosophy of generic programming, we do not wish to be limited to a particular implementation of these objects, so our curved kernel is parameterized by a `BasicKernel` parameter and derives from it, in order to include all needed functionality on basic objects, e.g. the number type `RT` (discussed in Section 4) and the circle type `Circle_2`.

As will be clear in the sequel, the predicates and constructions make heavy use of algebraic operations. We want to be as independent as possible from a particular implementation of the algebraic operations, so our curved kernel is parameterized by an `AlgebraicKernel`.

The declaration is the following:

```
template < typename BasicKernel,
           typename AlgebraicKernel >
class Curved_kernel;
```

The Curved kernel can be used the following way, for example, in the case when a user wishes to use the CGAL Cartesian kernel with the `double` number type as `RT`, and his own algebraic kernel, denoted `My_Algebraic_kernel`:

```
#include <CGAL/Cartesian.h>
#include <My_Algebraic_kernel.h>
#include <ECG/Curved_kernel.h>

typedef CGAL::Cartesian<double>        BK;
typedef My_Algebraic_kernel<BK::RT>    AK;
typedef ECG::Curved_kernel<AK,BK>      CK;
```

The interface provided at the geometric level is composed of:

- Types defining the objects.

Some of them are inherited from the basic kernel, namely the number type (denoted as `RT` in the sequel), and basic geometric types such as points, circles, and conics.

Some other types are inherited from the algebraic kernel. These types are mainly types for algebraic numbers of degree up to 4 and bivariate polynomials of degree 2.

Finally, some types are defined by the curved kernel itself, mainly: `Circular_arc_2`, `Circular_arc_endpoint_2`, `Conic_arc_2`, `Conic_arc_endpoint_2`.

• Predicates defined on the above objects; their functionality is available through two interfaces:

The first is based on global (non-member) functions. For example, comparing the abscissae of 2 endpoints can be done as follows:

```
Conic_arc_endpoint_2 p, q;
...
if (less_x(p, q))
  ...
else
  ...
```

The second is a functor which is provided by the kernel (through a member function of the kernel). Functors are classes providing a function operator : `operator()`, which can be called using the calling syntax of normal functions, with the added feature that the functor object can store a state. Functors are mostly useful in *generic* implementations of algorithms, such as those provided by the STL. They can be used as in the following example, for sorting a list of conic arc endpoints:

```
CK ck;
...
std::list<Conic_arc_endpoint_2>  L;
...
// Sorts L according to the x-order :
L.sort(ck.less_x_2_object());
...
```

• Constructions are available through the same double interface as the predicates. For example, the construction that splits an arc at an endpoint (a previously generated intersection) in two arcs is:

```
Circular_arc_2 c1;
Circular_arc_endpoint_2 p;
...
std::pair<Circular_arc_2, Circular_arc_2> result;
result = split(c1, p);
```

3. GEOMETRIC OBJECTS

We discuss arcs of conic curves in the plane, since this is what we have currently implemented. We have thus provided the tools for CGAL to build arrangements, using both the incremental and the vertical sweep line algorithms. Our choices readily extend to higher degree algebraic curves. The representations and predicates below rely on the algebraic concepts of Section 4.

3.1 Representation

A conic curve is a *curve*, provided by the Curved kernel, represented by a bivariate polynomial of total degree 2, whose coefficients are of type `RT`. The current implementation assumes that this polynomial always contains at least one quadratic term, i.e., x^2, y^2 or xy. Future work shall extend the implementation in order to include straight lines. On the other hand, curves can be generalized to algebraic (or implicit) curves of higher degree.

Two kinds of points —intersection points and endpoints— are considered in the same way, thus allowing us to have a unique representation. We call *endpoint* indifferently an endpoint of an arc or an intersection. An *endpoint* is represented by the two intersecting conics and its coordinates. In the current implementation, these are algebraic numbers of degree at most 4, represented by the *RootOf_d* concept.

This representation is redundant, since the coordinates could be retrieved at any time by a computation on the input curves, but avoiding to recompute them each time they are needed saves time. Moreover, keeping the original curves that define the points allows to perform some kind of elementary geometric filtering of predicates: for instance, to test the equality of two endpoints, we can first compare the curves defining them; computations on the coordinates will be performed only if the defining curves are not the same.

A conic *arc* is represented by a supporting conic, the two delimiting endpoints and a boolean indicating whether it lies on the upper or lower part of the conic curve. The input of course allows for full curves as well as arbitrary arcs, including non-monotone arcs. These are broken into x-monotone arcs (see **make_x_monotone** below) by the CGAL arrangements algorithms that fix the choice of axes and orientation.

3.2 Main primitives

We now present the main predicates and constructions for conic arcs that are required by the CGAL arrangements and that are provided by the curved kernel. In the case of circles, these primitives are simplified.

As will be clear in the description below, each of these geometric operations of the curved kernel calls predicates or constructions on purely algebraic objects, namely the algebraic numbers and the bivariate polynomials of the algebraic kernel. These algebraic primitives are described in Section 4.

make_x_monotone. It subdivides the given arc (which may be an entire curve) into x-monotone arcs. It calls `x_critical_points` on the equation of the curve, then creates the correct representations of all arcs.

nearest_intersection_to_right. Given an endpoint p and two x-monotone arcs, find their first intersection to the right of p. It computes all intersections of the two supporting curves by calling the function `solve` on the equations of the curves, then applies the `operator<` on the algebraic numbers representing the x-coordinates of the intersection points and of p.

compare_y_to_right. Given two x-monotone arcs supported by the conics g_1, g_2, and one of their intersection points $p = (p_x, p_y)$, such that the arcs are defined to the right of p (i.e. for x larger than p_x), the predicate decides which arc is above immediately to the right of p. If p is not defined as an intersection of g_1, g_2, the predicate expresses it as such by calling `solve`. The goal is to have a rational isolating interval (t, s) for p_x. If p is not the rightmost intersection of g_1, g_2, the predicate compares the y-roots of $g_1(s, y), g_2(s, y)$ by `operator<`. Otherwise, it calls **compare_y_at_x** on the right endpoint of an input arc with the smallest abscissa and on the other input arc.

In the case of circles, the predicate is simplified by using the slopes of the tangents at p.

compare_y_at_x. It decides whether a given arc is above or below a given endpoint $\gamma = (\gamma_x, \gamma_y)$, by calling `sign_at` with arguments the polynomial defining the curve supporting the arc, γ_x, and γ_y. Then, multiply the result by ± 1 depending on whether the given arc is on the upper or lower part of its curve. Future optimizations include techniques to use the knowledge of the polynomials defining the curves intersecting at γ.

In the case of a circular arc, the predicate simplifies by using the center.

4. ALGEBRAIC KERNEL CONCEPT

In the C++/STL sense, a concept is a set of requirements that a type must provide in order to be usable by some template function or class.

The algebraic kernel appearing as template parameter `AlgebraicKernel` of the curved kernel `Curved_kernel` (see Section 2) is supposed to provide the latter with algebraic numbers of degree up to 4 and operations on uni- or bivariate polynomials of degree 2. The two corresponding concepts are detailed in this section and our implementation of them discussed in the next section.

4.1 Real algebraic numbers

Most geometric predicates on circular arcs (resp. conic arcs) can be expressed as comparisons involving roots of degree 2 (resp. 4) polynomials. These polynomials are typically univariate resultants whose coefficients are themselves polynomials in the coefficients of the equations of the circles (resp. conics). Since there are several ways to deal with such algebraic numbers, we have tried to factorize as much code as possible between these alternatives. In C++, this means writing code that can act generically (through templates) on different types representing the numbers.

We differentiate here on the various categories of numbers using different C++ types, so that the representation of a number of degree 2, for instance, can be different from that of a number of degree less than 2 (the original curves' coefficients as well as quotients of these numbers), hence we have a set of concepts `RootOf_`d that describe basically the same requirements, for each degree d from 1 to 4. The bridge between the number type storing the curves' coefficients, supposed to be a *ring type*, named `RT` in the sequel, and the numbers of degree d is the `make_root_of_`d function described below.

4.1.1 Approaches.

Dealing with roots of polynomials can be done in several ways that generic code should support:

• approximate handling, using floating point approximation, e.g. C++ `double`.

• approximate but certified handling using interval arithmetic, for instance `CGAL::Interval_nt` or `boost::interval`.

For these first two cases, algebraic numbers of degree 2 can be implemented using the `sqrt()` function and the usual formula to produce an approximation. The corresponding *RootOf_2* type is then `RT` itself. For degrees higher than 2, algebraic numbers could be implemented using the Newton iteration, for instance.

• exact handling, using number types implementing exact comparisons between numbers built in the following ways:

– for degree 2: using the square root function `sqrt()`, e.g. `leda::real` or `CORE::Expr`.

– for degrees > 2: using the so-called *diamond operator* of `leda::real` [4] whose implementation is in progress, or the `CORE::rootOf` function of `CORE::Expr`, which manipulate roots of (currently only square-free) polynomials.

• polynomial representation of these roots using algebraic methods for comparisons [5, 8, 14]. We provide an implementation of this model, for algebraic numbers of degree up to 4, with the `Root_of` class described in Section 5.1, which has been integrated in the SYNAPS library. We also provide a specialized version, namely `Root_of_2`, restricted to degree 2, described in Section 5.2.

4.1.2 The RootOf_d concept.

Given a type `RT` representing the coefficients of the circles or conics equations, there must be a way to find out the (preferred) type used to store the roots of a degree ≤ 4 polynomial whose coefficients are of type `RT`. This type must also be the return type of the `make_root_of_`d function taking `RT` as the type of the arguments. This type must be defined in the algebraic kernel as `T_RootOf<d>::type`, ie. the `AlgebraicKernel` must contain:

```
typedef T_RootOf<d>::type RootOf_d;
```

The following functions must be defined:

• `RootOf_`d `make_root_of_`d`(RT` a_d`, ..., RT` a_0`, int i);`

which returns an object representing the i^{th} real root of the polynomial $P(X) = \sum_{j=0}^{d} a_j X^j$,

• `bool operator<(RootOf_`d `a, RootOf_`d `b);`
`bool operator<(RootOf_`d `a, RT b);`

which compare two algebraic numbers, or an algebraic number with a number of the ring type. Similarly, the other comparison operators (`>`, `<=`, `>=`, `==`, `!=`) have to be provided.

For all the possible models of this concept mentioned above, we have written the necessary functions, and now our kernel can be instantiated with all these number types (see Section 5).

4.2 Polynomials

As can be seen from Section 3.2, operations on *bivariate* polynomials play a crucial role in the evaluation of predicates. For now, the polynomials shall be of total degree 2 and denoted by Pol_2_2, but the concept can be generalized to arbitrary multivariate polynomials. More precisely, the functionalities required are:

•
```
template <class OutputIterator>
OutputIterator
x_critical_points(Pol_2_2 f, OutputIterator res);
```

which computes the pairs of elements of type *RootOf_d* that are the critical points of a polynomial f in the x direction, and copies them in an output iterator in lexicographical order.

•
```
template <class OutputIterator>
OutputIterator
solve(Pol_2_2 f1, Pol_2_2 f2, OutputIterator res);
```

which computes the pairs of elements of type *RootOf_d* that are the common roots of two polynomials $f1$ and $f2$, and copies them in an output iterator in lexicographical order.

• `int sign_at( Pol_2_2 f, RootOf_`d γ_x`, RootOf_`d γ_y`);`

which returns the sign of a bivariate polynomial $f(x, y)$ evaluated at algebraic numbers $x = \gamma_x$ and $y = \gamma_y$.

5. MODELS OF THE ALGEBRAIC KERNEL

Implementations matching the concepts are called models [1]. This section discusses models of the concepts related to the algebraic kernel presented above.

5.1 The `Root_of` class

Each algebraic number is represented by a polynomial and an index. When an algebraic number is constructed from a polynomial, we compute the discriminant of the polynomial in order to extract the multiple roots, if any, and to decide an isolating interval for this root. By this approach we have several advantages. The polynomial that represents an algebraic number is always square free and, moreover, any multiple root is represented by a lower degree algebraic number, which is rational in all cases except in the case of quartics with 2 double roots. The computation of the isolating intervals is achieved by close formulas using Lemma 1 and is independent of the separating bound of the roots.

Here we discuss polynomials of degree up to 4. Our methods should extend to degree 5 and, eventually, higher degrees. They are based on *static*, or precomputed, Sturm sequences as in [14, 7, 8]. Our algorithms test the sign of certain quantities, which are polynomials in the coefficients of the input bivariate equations representing the conics. These quantities are compiled in order to reduce execution time.

From a complexity viewpoint, we wish to minimize the degree of the tested quantities in these coefficients and, ultimately, the total number of operations as well. One originality of our approach is based on isolating the real roots by rational numbers, by applying the following fact.

LEMMA 1. *[18] Given a polynomial $P(X)$ with adjacent real roots γ_1, γ_2, and any two other polynomials $B(X), C(X)$, let $A(X) := B(X)P'(X) + C(X)P(X)$, where P' is the derivative of P. Then A or B have at least one real root in the closed interval $[\gamma_1, \gamma_2]$.*

Considering the polynomial remainder sequence of P and P', we can obtain, as a corollary, that $\deg A + \deg B \leq \deg P - 1$. The lemma is applied, as in [8], in order to specify several isolating polynomials, denoted by A or B in the lemma, of degree up to 2, when $\deg P \leq 4$. The roots of isolating polynomials constitute isolating points, since they isolate the roots of the original polynomial P. In the favorable cases, at least one isolating point is rational for every pair of consecutive real roots.

Even when this is not the case, it is possible to find rational isolating points by considering two or more isolating quadratic algebraic numbers in the interval defined by two roots. In short, for every polynomial of degree up to 4, our implementation provides rational isolating points between any root pair [8].

The maximum algebraic degree of any polynomial in the Sturm sequence is the degree of the resultant of the two polynomials, which is 8 when the inputs are quartics. In order to bound the maximum algebraic degree involved in the comparison of the roots of two quartics we must also consider the evaluation of the Sturm sequences on the endpoints of the isolating intervals of the roots.

PROPOSITION 1. [8] *There is an algorithm that compares any two roots of two quartics using Sturm sequences and isolating intervals from lemma 1, and the algebraic degree of the quantities involved is between 8 and 13.*

We have implemented static Sturm sequences, including all degenerate cases, i.e. when the degree of one of the input or intermediate polynomials is smaller than its nominal degree. In order to simplify this task we consider an evaluation scheme for the complete Sturm sequence. We can treat all possible evaluations of the Sturm sequence as a binary tree, which has as nodes the evaluation of a term of the sequence and branches according to the sign of the computed quantity. This algorithm is automatically generated.

In conclusion we obtain the `Root_of` class, constructed by a polynomial and an index. It includes the computation of an isolating interval for the algebraic number. Additionally, we have implemented the function
`sturm(Poly<RT> f, Poly<RT> g, RT a, RT b)`
which computes the result of the evaluation of the Sturm sequence of f and g, which are of arbitrary degree over an interval $[a, b]$. We do this computation statically for degree up to to 4 and dynamically (using various well known algorithms) for higher degrees.

Based on the aforementioned tools, Sturm theory [22, 8] yields the following functionalities. They are in the current release of the SYNAPS library.

- `compare(Root_of` α`, Root_of` β`)`

  Comparison of 2 algebraic numbers of degree ≤ 4.

- `sign_at(Poly` f`, Root_of` α`)`

  Determination of the sign of a univariate polynomial of arbitrary degree, evaluated over an algebraic number of degree up to 4.

5.2 Implementations of the `Root_of_2` class

In order to handle circles efficiently, we also provide another concrete model optimized for degree 2 – compared to the more general `Root_of` – which is using the following internal representation:

• three coefficients of type `RT` specifying the polynomial of degree 2.

• one boolean value specifying whether the smaller of the roots is considered or the other.

This class uses specific algebraic methods, based on resultants and Descartes' rule of sign, in order to handle comparisons [5]. Handling degrees higher than 2 is not implemented in this model, so it only handles circles. Our experiments below show that the gain of this specific technique is small. But the main reason for adopting a generic approach in the implementation of `root_of` has to do with programming effort: if we had extended the `Root_of_2` class and had specialized classes for every degree, we would have to develop a large amount of code, in order to handle all the situations efficiently.

Another idea that we have used to speed up computations substantially when `RT` is a rational type, such as `CGAL::Quotient` or multi-precision rational numbers such as those provided by GMP,[6] is to consider the polynomials with coefficients over the *ring type* defining the numerator and denominator of the rationals, instead of the rationals themselves, by simply multiplying the polynomial equation by all denominators; there are three denominators, since we are dealing with quadratic polynomials. This way, when we manipulate the coefficients of the polynomials, we manipulate multi-precision integers instead of rationals, which prevents the explosion of these numbers and is faster in general.

5.3 The Pol_2_2 class

We describe here the implemented methods which offer the functionalities required on bivariate polynomials of total degree up to 2. Although our tests in the next section are limited to the case of circles, we show below that extending them to arbitrary conics is straightforward. Our algorithms have been integrated, as a separate module, in the library SYNAPS 2.1.

In what follows an algebraic number γ is of degree up to 4 and it is represented by a polynomial R_γ and an isolating interval $I_\gamma = [a_\gamma, b_\gamma]$.

• `x_critical_points(Pol_2_2` f`)`

Take the derivatives f_x, f_y with respect to x and y. In order to specify the abscissae (resp. ordinates) of the critical points, we take the resultant of f and f_y (resp. f_x) by eliminating y (resp. x). To find the correspondence between x and y coordinates, we consider the slope of the line $f_y = 0$. E.g., if the slope is negative, then the biggest root of R_x corresponds to the biggest root of R_y. If the slope is zero, then this means that $f_y = 0$ is a horizontal line and R_y has one double (hence rational) root.

• `solve(Pol_2_2` f_1`, Pol_2_2` f_2`)`

We consider the resultants R_x, R_y of f_1, f_2 by eliminating y and x respectively, thus obtaining degree-4 polynomials in x and y. The isolating points of the resultants define a grid of boxes, where the intersection points are located. The grid has 1 to 4 rows and 1 to 4 columns; each box contains a simple or multiple root. It remains to decide, for certain boxes, whether they are empty and, if not, whether they contain a simple or multiple root. All multiple roots of the two resultants are either rational or quadratic algebraic numbers. Hence, several cases can be decided by calling `sign_at` with algebraic numbers of degree up to 2.

The rest of the cases reduce to testing if a box is empty, provided it contains at most one intersection point. We can decide the if a box is empty or not by calls to the `sign_at` function. Notice that the number of intersection points in a column (row) can not exceed the multiplicity of the algebraic number. Since the edges of the boxes are computed by the isolating points of the resultants, we can always guarantee that they contain at most one.

Unlike [3], where the boxes cannot contain any critical points of the intersecting conics, our algorithm does not make any such assumption about the topology of the conics in the boxes, hence there is no need to refine them.

Our approach can be extended in order to compute intersection points of curves of arbitrary degree, provided that we obtain isolating points for the roots of the two resultants, either statically (as above) or dynamically. In generalizing to higher degree, one would compare with existing work, such as [6].

• `sign_at(Pol_2_2` f`, Root_of` γ_x`, Root_of` γ_y`)`

We consider $f(X,Y)$ as a univariate polynomial with respect to X, and we compute the Sturm-Habicht sequence of R_{γ_x} and f. In order to find the sign of f evaluated over the algebraic number γ_x, we have to evaluate the sequence over the endpoints of the isolating interval of γ_x and count the modified sign changes [2]. The polynomials in this sequence are in two variables, in general. To be more precise, they are all in two variables, except the first one which is a univariate polynomial in X and the last one which is a univariate polynomial in Y. When we evaluate the sequence over the endpoints of the isolating interval of γ_x, the polynomials become either constant numbers (this is the case of the first polynomial) or univariate polynomials in Y. Therefore, we have to compute the sign of each polynomial in the sequence, evaluated over the algebraic number γ_y. This can be done by calling `sign_at(Poly` S_i`, Root_of` γ_y`)`, where S_i is a polynomial in the sequence, after the evaluation over an endpoint of the isolating interval of γ_x. Since the Sturm-Habicht sequences have nice specialization properties, the above procedure works for every specialization of the bivariate quadratic function f. The polynomial with the highest degree is the last one in the sequence, which is of degree 8, with respect to Y.

In order to reduce further the required arithmetic operations, we implemented the computation of the bivariate Sturm-Habicht sequence in `MAPLE` and we used package `codegeneration[optimize]`, so as to identify common subexpressions. By this technique we reduce the arithmetic operations to 1/3. The entire algorithm for `sign_at` is sketched, in preliminary form, in [8].

6. EXPERIMENTAL RESULTS

6.1 Traits class for CGAL arrangements

The CGAL arrangement package does not use the *kernel traits* design where predicates and constructions are defined as *objects*. Instead, it requires that the user gives a *traits class* with member functions providing the algorithms with those functionalities. Therefore, we wrote a traits class to interface our curved kernel with the arrangement algorithms. This traits class follows the requirements presented in [9] for both sweep line and incremental[7] algorithms.

6.2 Benchmarks

The hardware of our experiments was Pentium 4 at 2.5GHz with 1GB of memory, running Linux (2.4.20 kernel). The compiler was g++3.3.2; all configurations were compiled with the -DNDEBUG -O3 flags. The versions of the libraries are: CGAL 3.0, CORE 1.6x, GMP 4.1.2, LEDA 4.5.

The package used to compute the arrangement is the only publicly available implementation of arrangements, namely the CGAL implementation (sweep line version).

[6] http://www.swox.com/gmp/

[7] All predicates in the traits class are taken from the curved kernel except *curves_compare_y_at_x_to_right(c1,c2,p)* as used in the incremental setting: the case when p is not a common point of $c1$ and $c2$ is not considered in the kernel, and is currently evaluated approximately in the traits (since it has higher algebraic degree, and we believe that using it could be avoided).

The implementation of conic arcs is not finalized, hence not benchmarked. We show tests with 5 different kinds of inputs of circular arcs: Random full circles (RFC). Circles positioned on the cross-points and the centers of the cells of a square grid with cell size 10^4 (G). Configuration G, with every circle perturbed both in radius and center by a random integer in the range [-100,100) (PG). Random monotone arcs (RMA) generated by triplets of random circles (one is the supporting circle and the other two define the endpoints of the arc). The same as RMA, except that arcs may be non-monotone (RNMA). The input size refers to the number of full circles for RFC, G, and PG, and to arcs for RMA and RNMA. All random inputs were generated using random integers of 16 bits.

All experimental data are shown in table 1. The output size per data set is given in terms of endpoints and *half-edges*, where two half-edges correspond to each arc defined by the arrangement. “CK” indicates the use of our Curved kernel. Lines marked “CGAL” correspond to the conic traits from CGAL. Timings in seconds are given in the last 5 columns.

`MP_Float` is a number type from CGAL which computes polynomial operations $(+, -, \times, <)$ exactly over floating point values, `Quotient` is a rational template type from CGAL, `mpz_class` and `mpq_class` are GMP types, `Lazy` is `CGAL::Lazy_exact_nt`, a template number type doing lazy evaluation, adapted to wrap our `Root_of_2` type. `Root_of_2` is the type described in section 5.2, while `Root_of2` is a specialized version for degree two of the type described in section 5.1. Dashes indicate that the corresponding test could not be completed mainly due to insufficient memory; this occurs with CORE and `Lazy`, the reason being the high precision needed or a large expression tree constructed in conjunction with the large number of coordinates constructed.

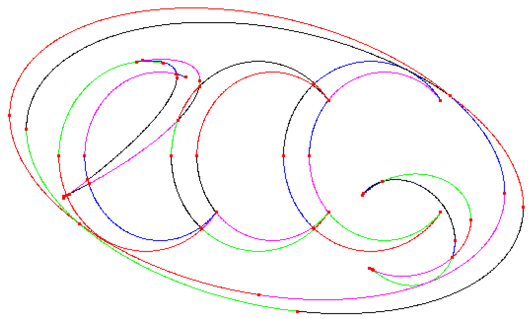
We conclude that `Root_of_2` is most efficient with the `mpz_class` type, however the configuration with `mpq_class` is more general since it can treat rational data. The performance of `Root_of_2` as compared to `Root_of2` is the same for all number types tested, hence we do not use all number types with `Root_of2`. Our tuned arithmetic performs better than the more general CORE (that sometimes aborts due to memory consumption) and LEDA arithmetic, except for the G configuration, where all intersections are integer points.

Compared to the CGAL traits with LEDA arithmetic, our code is much faster except from the G (grid) configuration, where the two show the same performance. The reason is that the CGAL traits use bounding boxes around the conic arcs, and this proves particularly useful in the G configuration. The CGAL traits can handle conic arcs *in general*, while our benchmarks currently handle only circular arcs. Our code for conic arcs is still in preliminary form.

One should note that the G and PG distributions are more easily treated by the algorithm, due to fewer intersections and, in the case of G, due also to lower arithmetic precision required. On the other hand, the random cases yield more intersections, hence a larger computational effort is needed.

6.3 Demo

We implemented a demo that was used to produce both pictures below, the first displaying the acronym ECG. The demo uses Qt.[8] It allows to compute an arrangement of elliptic arcs with CGAL, either incrementally or with the sweep algorithm. Then it displays the computed arrangement as a set of arcs, together with the endpoints, which are either x-extremal points of each arc, or intersection points between two arcs, or the endpoints of the original arcs.

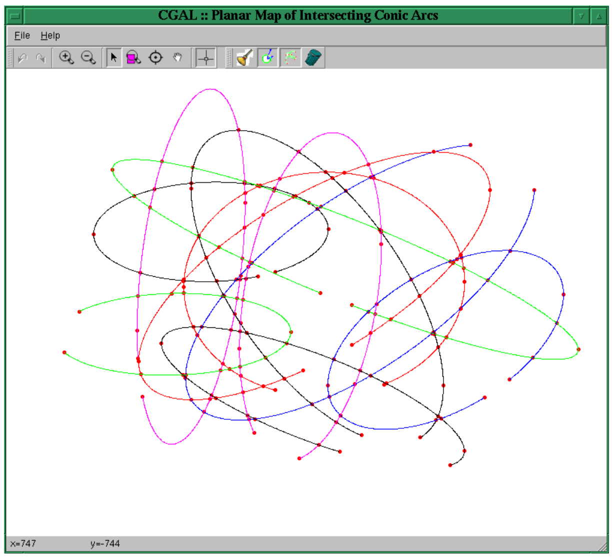


7. FUTURE WORK

The work described in this paper will be submitted for integration into the CGAL Open Source Library. Our curved kernel now contains a solid implementation for arrangements of circular arcs. The polishing of the code for conic arcs is in progress.

The work in [7, 8] is being extended so as to implement algebraic numbers of arbitrary degree in SYNAPS, which will allow us to deal with implicit curves of arbitrary degree. The algorithms, based on Sturm sequences, have certain similarities with the corresponding class of algorithms implemented in [12]. The latter discusses other approaches, as well. We are currently investigating ways of making our implementations benefit from each other.

Lastly, we are going to study the use of more filtering techniques in the manipulation of algebraic numbers for geometric predicates, which should lead to an improved efficiency.



[8] http://www.trolltech.com/

Arcs	Traits	NumberType	AlgebraicType	RFC	G	PG	RMA	RNMA
150	CK	MP_Float	Root_of_2<MP_Float>	39	1.4	1.8	2.2	7.0
150	CK	CGAL::Quotient<MP_Float>	Root_of_2<MP_Float>	56	2.0	2.6	3.2	10
150	CK	::mpz_class	Root_of_2<::mpz_class>	40	1.4	1.7	2.3	7.1
150	CK	::mpz_class	Root_of2<::mpz_class>	40	1.4	1.7	2.2	7.0
150	CK	::mpq_class	Root_of_2<::mpz_class>	61	2.0	2.5	3.5	12
150	CK	CGAL::Lazy<mpz_class>	CGAL::Lazy<Root_of_2<mpz_class>>	73	2.0	2.8	3.5	13
150	CK	CORE::Expr	CORE::Expr	193	5.8	6.0	5.1	28
150	CK	LEDA::real	LEDA::real	98.5	2.59	3.96	4.60	20.5
150	CGAL	LEDA::real	LEDA::real	56.8	1.32	3.15	2.33	13.4
150			Vertices:	10340	300	572	713	2275
			Half-edges:	40760	1148	1688	1952	7868
300	CK	MP_Float	Root_of_2<MP_Float>	137	2	5	10	34
300	CK	CGAL::Quotient<MP_Float>	Root_of_2<MP_Float>	195	3	7	14	51
300	CK	::mpz_class	Root_of_2<::mpz_class>	137	2	5	10	34
300	CK	::mpz_class	Root_of2<::mpz_class>	137	2	5	10	34
300	CK	::mpq_class	Root_of_2<::mpz_class>	216	3	7	15	58
300	CK	CGAL::Lazy<mpz_class>	CGAL::Lazy<Root_of_2<mpz_class>>	275	3	8	15	64
300	CK	CORE::Expr	CORE::Expr	-	10	17	20	195
300	CK	LEDA::real	LEDA::real	374	5.18	11.4	19.0	104
300	CGAL	LEDA::real	LEDA::real	410	2.62	8.01	10.1	98.3
300			Vertices:	34890	632	1412	2202	10157
			Half-edges:	138360	2360	4448	7008	38218
500	CK	MP_Float	Root_of_2<MP_Float>	374	6	7	20	86
500	CK	CGAL::Quotient<MP_Float>	Root_of_2<MP_Float>	531	8	10	28	128
500	CK	::mpz_class	Root_of_2<::mpz_class>	372	5	7	20	87
500	CK	::mpz_class	Root_of2<::mpz_class>	372	5	7	20	87
500	CK	::mpq_class	Root_of_2<::mpz_class>	587	8	10	31	146
500	CK	CGAL::Lazy<mpz_class>	CGAL::Lazy<Root_of_2<mpz_class>>	-	8	11	33	167
500	CK	CORE::Expr	CORE::Expr	-	23	23	57	-
500	CK	LEDA::real	LEDA::real	1143	10.7	14.9	46.2	257
500	CGAL	LEDA::real	LEDA::real	2689	5.12	11.8	33.0	395
500			Vertices:	92664	1000	1952	4967	24819
			Half-edges:	368656	3908	5808	16868	95276

Table 1. Experimental results

Acknowledgments

The authors acknowledge constructive discussions with Lutz Kettner and Bernard Mourrain. They are grateful to Radu Ursu for his help on the graphical interface.

References

[1] M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.

[2] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in real algebraic geometry*. Springer-Verlag, Berlin, 2003.

[3] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *Proc. 10th European Symposium on Algorithms*, volume 2461 of *Lecture Notes Comput. Sci.*, pages 174–186, 2002.

[4] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. In F. Meyer auf der Heide, editor, *Proc. 9th European Symposium on Algorithms*, volume 2161 of *Lecture Notes Comput. Sci.*, pages 254–265, 2001.

[5] O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comput. Geom. Theory Appl.*, 22:119–142, 2002.

[6] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact, and efficient computations with cubic curves. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, 2004.

[7] I.Z. Emiris and E.P. Tsigaridas. Methods to compare real roots of polynomials of small degree. Technical Report ECG-TR-242200-01, INRIA Sophia-Antipolis, 2003.

[8] I.Z. Emiris and E.P. Tsigaridas. Comparison of fourth-degree algebraic numbers and applications to geometric predicates. Technical Report ECG-TR-302206-03, INRIA Sophia-Antipolis, 2003. Final version submitted for publication.

[9] E. Fogel, D. Halperin, R. Wein, M. Teillaud, E. Berberich, A. Eigenwillig, S. Hert, and L. Kettner. Specification of the traits classes for cgal arrangements of curves. Technical Report ECG-TR-241200-01, INRIA Sophia-Antipolis, 2003.

[10] G. Gatellier, B. Mourrain, and J.-P. Pavone. Axel: Algebraic software component for geometric modeling, 2003. INRIA Sophia-Antipolis, Manuscript.

[11] N. Geismann, M. Hemmer, and E. Schömer. Computing a 3-dimensional cell in an arrangement of quadrics: Exactly and actually! In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 264–273, 2001.

[12] L. Guibas, M. Karavelas, and D. Russel. A computation framework for handling motion. In *Proc. ALENEX*, 2004. To appear.

[13] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes Comput. Sci.*, pages 79–90. Springer-Verlag, 2001.

[14] M. I. Karavelas and I. Z. Emiris. Root comparison techniques applied to computing the additively weighted Voronoi diagram. In *Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 320–329, 2003. Final version to appear in *Comp. Geometry: Theory & Appl.*.

[15] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2001.

[16] B. Mourrain, J.-P. Técourt, and M. Teillaud. Predicates for the sweeping of an arrangement of quadrics in 3d. Technical Report ECG-TR-242205-01, INRIA Sophia-Antipolis, 2003.

[17] S. Pion and M. Teillaud. Towards a cgal-like kernel for curves. Technical Report ECG-TR-302206-01, MPI Saarbrücken, INRIA Sophia-Antipolis, 2003.

[18] T. Sederberg and G.-Z. Chang. Isolating the real roots of polynomials using isolator polynomials. In C. Bajaj, editor, *Algebraic Geometry and Applications, Spec. Issue of Symp. on occasion of S. Abhyankar's 60th Birthday*. Springer Verlag, 1993.

[19] R. Wein. High-level filtering for arrangements of conic arcs. In *Proc. 10th European Symposium on Algorithms*, volume 2461 of *Lecture Notes Comput. Sci.*, pages 884–895, 2002.

[20] R. Wein. Cgal based implementation of arrangements of conic arcs. Technical Report ECG-TR-241210-01, Tel-Aviv University, 2003.

[21] N. Wolpert. Jacobi curves: Computing the exact topology of non-singular algebraic curves. In G. D. Battista and U. Zwick, editors, *Proc. 11th European Symp. Algorithms*, volume 2832 of *Lecture Notes Comput. Sci.*, pages 532–543, 2003.

[22] C. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, New York, 2000.