Exact and efficient computations on circles in CGAL and applications to VLSI design
Pedro M. M. de Castro, Sylvain Pion, Monique Teillaud

To cite this version:

HAL Id: inria-00123259
https://inria.hal.science/inria-00123259v2
Submitted on 11 Jan 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Exact and efficient computations on circles in CGAL and applications to VLSI design

Pedro M. M. de Castro — Sylvain Pion — Monique Teillaud

N° 6091
Janvier 2007

Thème SYM
Exact and efficient computations on circles in CGAL
and applications to VLSI design

Pedro M. M. de Castro†, Sylvain Pion‡, Monique Teillaud††

Thème SYM — Systèmes symboliques
Projets Geometrica

Rapport de recherche n° 6091 — Janvier 2007 — 14 pages

Abstract: CGAL (Computational Geometry Algorithms Library) is a large collection of geometric objects, data
structures and algorithms. CGAL currently offers mostly functionalities for linear objects (points, segments,
lines, triangles...).

The first version of a kernel for circles and circular arcs in 2D was recently released in CGAL 3.2. We show
in this paper a variety of techniques that we tested to improve the efficiency of the 2D circular kernel. These
improvements are motivated by applications to VLSI design, and real VLSI industrial data are used to measure
the impact of the techniques used to enhance this kernel. The improvements will be integrated in CGAL 3.3.

Key-words: implementation, effective geometry, algebraic numbers, curves

This work is partially supported by the IST Programme of the 6th Framework Programme of the EU as a STREP (FET Open
Scheme) Project under Contract No IST-006413 - ACS (Algorithms for Complex Shapes with certified topology and numerics) -
http://acs.cs.ru.nl/

* This work was done while Pedro M. M. de Castro was working at INRIA. Current address: Center of Computer Sciences,
Universidade Federal de Pernambuco, Brazil. pmmc@cin.ufpe.br
† Sylvain.Pion@sophia.inria.fr - http://www.sop.inria.fr/geometrie/team/Sylvain.Pion
‡ Monique.Teillaud@sophia.inria.fr - http://www.sop.inria.fr/geometrie/team/Monique.Teillaud

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65
Calculs exacts et efficaces sur des cercles dans CGAL et applications à la conception de circuits imprimés

Résumé : La bibliothèque CGAL (Computational Geometry Algorithms Library) offre une large collection d’objets, de structures de données et d’algorithmes géométriques. CGAL contient actuellement essentiellement des fonctionnalités pour des objets linéaires (points, segments, droites, triangles...).

La première version d’un noyau pour les cercles et les arcs de cercles en dimension 2 a été récemment incluse dans CGAL 3.2. Nous montrons dans ce papier des techniques variées qui ont été testées pour améliorer l’efficacité du noyau circulaire. Ces améliorations sont motivées par des applications à la conception de circuits imprimés, et l’impact des techniques est mesuré sur des données industrielles réelles. Les améliorations seront intégrées à CGAL 3.3.

Mots-clés : implantation, géométrie effective, nombres algébriques, courbes
1 Introduction

The goal of the CGAL Open Source Project is to provide easy access to efficient and reliable geometric algorithms to users in industry and academia. This is achieved in the form of the C++ Computational Geometry Algorithms Library [2]. The CGAL packages adopt the generic programming paradigm [19], making extensive use of C++ class templates and function templates, and their design is heavily inspired from the C++ Standard Template Library [5].

For instance, let us consider the case of geometric arrangements: an arrangement of a finite set of curves in the plane is the partition of the plane into faces, edges and vertices, that is induced by these curves [22] [6] [18]. A generic implementation of a data structure that handles arrangements is achieved by the CGAL::Arrangement_2 class:

\[ \text{template } <\text{class Traits}> \text{ class Arrangement}_2 \]

This class must be instantiated with a class, referred to as a traits class [28], that must define a type representing a certain family of curves, and some functions operating on curves of this family.

The CGAL kernels provide the users with basic geometric objects and basic functionalities on them. CGAL currently offers kernels for linear objects (points, segments, lines, triangles, ...), and the first version of a kernel for circles and circular arcs in 2D, called 2D circular kernel in the sequel, was recently released in CGAL 3.2 [29]. A kernel can be wrapped into a traits class offering the interface for some geometric algorithms; this was done for the CGAL circular kernel and CGAL::Arrangement_2. However, a kernel is meant to offer general purpose functionalities, while a traits class offers the minimum set of functionalities required by a specific class.

Robustness, achieved through the exact geometric computation paradigm [33], is probably the first strength of CGAL. The CGAL arrangement package, completely redesigned for CGAL 3.2 [22] [31] offers a robust and efficient implementation. Arrangements are used as fundamental structures for many applications, for example boolean operations in CAD/CAM and VLSI design [20]. ESOOLID [25], though based on exact computation, may crash on degenerate input data. Most industrial software compute these operations approximately, hence are not robust; to our knowledge, the only commercial software which provides exact boolean operations on polygons with circular arcs is LEDA [9]. Real VLSI data sets consist of line segments and circular arcs, containing many degenerate, or close to degenerate, cases (junctions, identical intersection points, tangencies...) requiring highly robust code.

Efficiency is also a crucial quality for the use of CGAL on real industrial data. VLSI inputs consist of very large data sets, which leads to the need for improvements in the efficiency of the 2D circular kernel. We show in this paper a variety of techniques from different nature that we tested to improve the 2D circular kernel, and experimental evidence of their impact is studied by benchmarking on real VLSI industrial data. The techniques resulting in measurable improvements will be integrated in CGAL 3.3.

Section 2 quickly presents the 2D circular kernel design, and Section 2.2 focuses on the basics on number types. The experimental framework is settled in Section 3. In Section 4 a first set of techniques for improvements is studied: bit-fields, number types optimizations, caching and reference counting. Section 5.1 is devoted to the representation of algebraic numbers, and Section 5.2 to geometric filtering.

2 The 2D Circular Kernel

To answer the need for robustness on manipulations of circular objects, a first version of the 2D circular kernel was released in CGAL 3.2. We describe in this section its more relevant characteristics. We also show its weaknesses supported by experimental evidence. Indeed, for the first release of the circular kernel, the focus was put on the design and the robustness of the package.

2.1 Design

The design of the 2D circular kernel [15] uses the extensibility and adaptability properties of the CGAL linear kernel [25]. The circular kernel is parameterized by, and inherits from, a LinearKernel parameter, for objects like points, circles and number types. It has a second parameter, AlgebraicKernel, providing algebraic operations that are necessary for computations on circles.

\[ \text{template } <\text{class LinearKernel, class AlgebraicKernel}> \text{ class Circular_kernel}_2 : \text{public LinearKernel} \]

\[ ^{1} \text{see } \text{http://research.cs.tamu.edu/keyser/geom/esolid/} \]
In this way, the circular kernel is limited neither to any particular linear kernel implementation, nor to any particular implementation of algebraic operations.

The geometric level interface provides types already defined by the linear kernel, plus three new types: `Circular_arc_point_2` for points on circles, and `Circular_arc_2` and `Line_arc_2` respectively representing circular arcs and line segments delimited by such points. It also provides several predicates and constructions, defined on objects of those types, as functors \[\mathbb{R}\] or global functions. A *predicate* is a test function returning a discrete set of values (for instance: “do the two curves \(c\) and \(c'\) intersect?”). A *construction* is a function that constructs new objects (for instance: “compute the intersection between \(c\) and \(c'\)”).

In this first release, predicates and constructions were implemented in order to meet at least all the requirements of the arrangement package. See \[29\] for more details.

2.2 Number Type

Standard computations on linear geometric objects can be performed using only comparisons of polynomial expressions in the basic number type used for the representation of objects. This can be performed as soon as the number type RT is a ring type, meaning that it offers the following operations: addition, subtraction, multiplication.

A circle in CGAL is known by the coordinates of its center and its squared radius, supposed to be represented by some ring number type. A line is known by its equation, supposed to have coefficients in the same ring type.

Intersections involving circles lead to manipulating algebraic numbers of degree two on this ring. Moreover, most geometric predicates on circular arcs are expressed as comparisons involving such roots. Dealing with roots of polynomials can be done in several ways that generic code should support: approximate handling (using e.g. C++ double); approximate but certified handling using interval arithmetic \[10\] (e.g. CGAL::Interval_nt\[\mathbb{R}\]) or boost::interval \[11\]); exact handling using separation bounds to implement exact comparisons between numbers (as in CORE \[3\] or LEDA \[5\], used in the EXACUS prototype library \[9\]), see \[27\] for a survey; and exact handling using polynomial representation of these roots and algebraic methods (like resultants and Descarte's rule of sign) which reduce the computations to comparisons of polynomial expressions \[18\ 21\ 13\ 17\].

It was shown that the latter choice, for this specific small degree two, was more efficient than using a general library like CORE or LEDA \[13\ 17\], so this choice was made in the implementation of the Algebraic_kernel released in CGAL 3.2 as a model for the AlgebraicKernel parameter. A template class Root_of_2<RT> is provided for algebraic numbers of degree 2, 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 root.

Besides, when the input is given by a rational type, such as any multi-precision rational (e.g. GMP rationals \[3\] or CGAL::Quotient<CGAL::MP_Float\[\mathbb{R}\]), we 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 denominators.

2.3 Kernel and traits class

The CGAL arrangement package comes with a default traits class for line segments and circular arcs, called Default_traits in the sequel. We wrapped the circular kernel into a traits class offering the same interface. As mentioned in the introduction, kernel and traits classes have important differences. We mention here two main distinctions:

- A traits class is tuned for a specific use. Since the arrangement package requires the traits to provide a unique type Curve_2, the default traits class does not offer separate types for line segments and circular arcs.

The circular kernel, meant to be general purpose, offers two different types, Circular_arc_2 and Line_arc_2. These distinguished types are available directly for the user. Moreover this choice will allow a generalization to a kernel for more complex curves in the future. The traits class built on the circular kernel uses the boost::variant class\[14 \ 11\], that allows to wrap the two complex types in one unique type Curve_2, in a similar way as a C++ union\[14\] does for basic types.

\[2\] Interval number type from CGAL
\[3\] Thematic Mapping Library Reference manual, several of these approaches \[29\].
\[4\] exact Multi Precision Float from CGAL
\[5\] http://www.boost.org/libs/variant/index.html
Figure 1: An example of VLSI data (vlsi_7), composed of 22,406 polygons and 294 circles, for a total number of 89,918 arcs. The zooms show the complexity of the data. The bottom picture shows the two synthetic data sets. The tables give the characteristics of the data sets: $N$ is the number of arcs (line segments or circular arcs), $V$ is the number of vertices of the arrangement, $E$ is the number of edges and $F$ the number of faces.

- Also, the set of functionalities in the traits is minimal. Arrangements algorithms implemented in this package start by cutting all curves into $x$-monotone arcs. So, in fact functionalities like intersection computations are provided only for $x$-monotone arcs in a traits class for arrangements. The circular kernel implements these functionalities for general arcs.

### 3 VLSI Data Sets and Conditions of Experiments

We conducted experiments to evaluate the practical influence of several improvement techniques on the CGAL 2D circular kernel. Our experiments consist in computing arrangements of real industrial data of VLSI models representing electronic circuits, with the efficient sweep-line algorithm provided by the new CGAL arrangement package.

The VLSI data have many cases of junctions, degeneracies and tangencies, causing approximate computation to fail due to rounding errors, and which make them appropriate for exact computation. See Figure 1 for an illustration. The table gives the sizes of the input files we are using for the experiments, together with the sizes of the corresponding arrangements.

We complete our experiments with synthetic input (see also Figure 1): a very dense distribution where circles are centered on a grid and all pairs of circles intersect (a), and a sparse distribution without intersection (b).

The hardware of our experiments was a Pentium 4 at 2.5 GHz with 1GB of memory, running Linux (2.4.20 Kernel). The compiler was g++ 4.0.2; all configurations were compiled with the -DNDEBUG -O2 flags.

In the sequel, the default traits class Default_traits will be used as a reference for measuring the performances of the circular kernel. Both of them are used with the same basic exact ring number type: CGAL::MP_Float.

<table>
<thead>
<tr>
<th>VLSI</th>
<th>N</th>
<th>V</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlsi_1</td>
<td>11,929</td>
<td>20,649</td>
<td>26,468</td>
<td>6,385</td>
</tr>
<tr>
<td>vlsi_2</td>
<td>9,663</td>
<td>8,556</td>
<td>9,439</td>
<td>887</td>
</tr>
<tr>
<td>vlsi_3</td>
<td>35,449</td>
<td>101,758</td>
<td>163,316</td>
<td>61,887</td>
</tr>
<tr>
<td>vlsi_4</td>
<td>12,937</td>
<td>81,096</td>
<td>143,049</td>
<td>61,986</td>
</tr>
<tr>
<td>vlsi_5</td>
<td>4,063</td>
<td>40,636</td>
<td>77,598</td>
<td>36,965</td>
</tr>
<tr>
<td>vlsi_6</td>
<td>74,052</td>
<td>547,250</td>
<td>1,016,460</td>
<td>470,480</td>
</tr>
<tr>
<td>vlsi_7</td>
<td>89,918</td>
<td>499,309</td>
<td>878,799</td>
<td>383,871</td>
</tr>
<tr>
<td>vlsi_8</td>
<td>123,175</td>
<td>370,304</td>
<td>555,412</td>
<td>190,031</td>
</tr>
<tr>
<td>vlsi_9</td>
<td>139,908</td>
<td>1,433,248</td>
<td>2,690,530</td>
<td>1,257,684</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distributions</th>
<th>N</th>
<th>V</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>very dense (a)</td>
<td>400</td>
<td>160,000</td>
<td>320,000</td>
<td>159,602</td>
</tr>
<tr>
<td>sparse (b)</td>
<td>441</td>
<td>882</td>
<td>882</td>
<td>442</td>
</tr>
</tbody>
</table>

\[\text{this explains the number of vertices, 882, in the sparse distribution (b) of Figure 1; each circle is cut into two half-circles, which results in two vertices and two edges in the arrangement.}\]
<table>
<thead>
<tr>
<th>Input</th>
<th>CGAL 3.2 circular kernel</th>
<th>Default_traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlsi_1</td>
<td>28.0</td>
<td>8.55</td>
</tr>
<tr>
<td>vlsi_2</td>
<td>48.0</td>
<td>2.59</td>
</tr>
<tr>
<td>vlsi_3</td>
<td>135</td>
<td>26.7</td>
</tr>
<tr>
<td>vlsi_4</td>
<td>569</td>
<td>26.9</td>
</tr>
<tr>
<td>vlsi_5</td>
<td>125</td>
<td>14.3</td>
</tr>
<tr>
<td>vlsi_6</td>
<td>611</td>
<td>137</td>
</tr>
<tr>
<td>vlsi_7</td>
<td>690</td>
<td>192</td>
</tr>
<tr>
<td>vlsi_8</td>
<td>3,650</td>
<td>220</td>
</tr>
<tr>
<td>vlsi_9</td>
<td>2,320</td>
<td>581</td>
</tr>
<tr>
<td>Very dense</td>
<td>335</td>
<td>77.9</td>
</tr>
<tr>
<td>Sparse</td>
<td>0.91</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 1: Time, in seconds, spent to compute the arrangement with the CGAL 3.2 circular kernel and Default_traits.

Note however that the default traits class is already optimized and uses arithmetic filtering (see Section 4.3.2) which makes it quite efficient. The CGAL 3.2 circular kernel is not yet filtered at all. Note also that the use of boost::variant (see Section 4.2) introduces a slight overhead in the running times obtained by the circular kernel.

These elements partly explain the poor performance of this kernel shown in Table 1. The rest of this work is devoted to show how several techniques can be combined to produce an important improvement in the running times.

4 Tuning the 2D Circular Kernel

In this section, we improve the circular kernel in various directions: bit-field, optimization of algebraic numbers, caching, and reference counting.

4.1 Caching Using Bit-Field

Geometric algorithms usually demand operations involving exact computation (not to be mistaken with exact arithmetics, see for instance Section 4.3.2). Nevertheless, the results of some costly operations can be stored to avoid recomputing them several times. When those operations return symbolic values, like boolean, the memory space used can be very low: the results can be efficiently stored in a bit-field which consists in the manipulation of individual bits of an entire block of data [4].

A Circular arc 2 is given by its supporting circle and its endpoints (given in counterclockwise order), that can have algebraic coordinates. The bit-field we introduce as an additional data member has 2 bytes (in fact, only 12 from the total 16 bits are used), and is declared in the following way:

```c
typedef struct bit_field {
    unsigned short int is_full:2; // is the arc a full circle?
    unsigned short int is_x_monotone:2; // is it a {x,y} monotone arc?
    unsigned short int is_y_monotone:2;
    unsigned short int is_complement_x_monotone:1; // is it the complement of a {x,y} monotone arc?
    unsigned short int is_complement_y_monotone:1;
    unsigned short int two_end_points_on_upper_part:2; // are the two endpoints on the {upper,left}
    unsigned short int two_end_points_on_left_part:2; // part of the circle ?
} bit_field;
```

Each one bit entry (e.g. is_complement_x_monotone) has two different states: true and false. The two bits entries (e.g. is_full, is_x_monotone) have an additional state: don't know; meaning that the operation was not yet computed. This third state is useful to make the computation only when necessary.

We already quickly mentioned that the arrangement algorithms start by cutting all input arcs into x-monotone arcs (Section 3). To this aim, a functor make_x_monotone must be provided by the traits class or the kernel. Any bit-field corresponds to a set of possible spatial configurations of the circular arc. Knowing at which point a given arc is cut allows to deduce the set of possible spatial configurations of the smaller arcs obtained. Then, when this set of configurations leads to a unique bit-field configuration, we can just get it in constant
time. In fact it is easy to check that cutting an arc into \( x \)-monotone arcs falls into this category of cut. Thus, the `make_x_monotone` function can set all the constructed arcs bit-fields with no need for any exact computation.

### 4.2 Caching Intersections of Supporting Circles

To compute the intersection of two circular arcs, we first compute the intersection of their respective supporting circles. In fact, in the VLSI input data, several circular arcs are often supported by the same circle. So, it sounds interesting to try to cache the intersections of circles, to avoid to recomputation them each time arcs contained in them are considered. We tried to use a `std::map` which takes a pair of circles and returns their (at most two) intersection points.

However, each query for a pair of supporting circles in the map takes a small amount of time to be answered, and when this answer is empty (because the intersection has not been computed yet), this query induces an overhead. Maps, like `std::map`, have an \( O(\log(n)) \) query time complexity and \( O(n) \) space complexity, where \( n \) is the number of pairs of supporting circles. Experiments on VLSI files showed that those maps incur a bigger variation on the running times, but keep the mean unchanged over all VLSI sets. Moreover, a memory overflow occurs on the biggest file `vlsi_9` (Figure 1). Since the industrial applicability of our work is crucial, we do not consider those maps as an interesting contribution to the enhancement of the circular kernel and drop them in the rest of this work.

### 4.3 Enhancing the Algebraic Number Type

The `Root_of_2` number type was improved by following two directions presented in this section.

#### 4.3.1 Optimizing Particular Cases

In some cases, the actual degree of the algebraic numbers computed is only one, i.e. these numbers are in fact rational numbers (e.g. for the intersection point of two tangent circles). However in CGAL 3.2, `Root_of_2` is not optimized for this special case, so, all operations on these numbers have the same complexity, whenever `Root_of_2` is a rational number or not.

We added a flag, `is_rational`, to the `Root_of_2` number type which indicates whether it is a rational number or not. This flag should be set to `true` whenever the number is identified as rational. All operations involving `Root_of_2` have special cases optimized according to the value of `is_rational`. Two constructors for `Root_of_2` are available, in order to avoid the user to set `is_rational` directly. The first constructor takes a `field` number type denoted as `FT`\(^8\) and sets `is_rational` automatically to `true`. The second takes the three coefficients `FT` of the polynomial (plus one additional boolean, specifying whether the algebraic number is the smallest root of the polynomial or not). It is up to the user to identify on their application those cases and use the right constructor in order to optimize their own code.

Algebraic kernel is the main target of this optimization, since it is responsible for creating `Root_of_2` numbers (see Section 2). Every time when the rationality of `Root_of_2` is detectable in constant time, we use the first constructor. Those cases appear
- in the intersection of two tangent circles;
- in the intersection of a line and a circle that are tangent;
- in the intersection of a vertical/horizontal line with a circle.

#### 4.3.2 Arithmetic Filtering

The general idea of filtering comparisons consists in computing an approximate result, together with an error bound. If the error bound is small enough, comparing the approximate values is enough to give the result of the comparison of the exact values, which allows to conclude very quickly. Otherwise, we say that the filter fails, and the computation is done using exact arithmetic. Clearly, the errors need to be kept as small as possible to avoid too many filter failures, since expensive exact computation must be used in that case, and the computations of error bound only induce an overhead \([10]\). The combination of exact computation and filtering techniques allows to get both fast and exact results.

We added such a filtering process to the `Root_of_2` number type as follows. The `to_interval()` function takes an exact `Root_of_2` number and returns an interval\(^9\) containing it. If the intervals containing two algebraic

---

\(^8\) a field type `FT` offers the three basic operations of a ring, plus the fourth basic operation of a field: division

\(^9\) in the form of a `CGAL::Interval_nt` number type
numbers do not overlap, we can just compare their extremities to know the result of the exact comparison of the numbers, otherwise the full exact comparison is performed 13.

As operations are performed on algebraic numbers, the coefficients of the polynomials representing them get bigger. The precision of the interval is directly related with the size of the coefficients. So, to reduce the number of filter failures, we simplify those coefficients before calling the to_interval() function. We tested two different possibilities:

- simplifying the coefficients at each Root_of_2 construction;
- simplifying the coefficients before computing the interval, inside the to_interval() function.

Our experiments show that the second option is faster. It is intuitive, since the constructor of Root_of_2 is called more times than the to_interval(). To simplify the coefficients in constant time, we must look deeper into the CGAL::MP_Float representation, which is $m2^{16e}$, where $m$ is the mantissa, represented as a vector of integers, and $e$ the exponent, represented as a double. Thus, dividing an MP_Float by $2^{16n}$ is a constant time operation. We simplify the coefficients by dividing each of them by $2^k$ in such a way that its exponent is closer to 0. We choose $k$ as the mean of the maximum and minimum exponents of the coefficients, which is a constant time operation as well, and works fine. Now, we can just use the usual formula solving degree 2 equations to obtain a good interval.

Another small optimization consists in handling special equations with a more appropriate formula. For example, we do not need to use $(-b \pm \sqrt{b^2 - 4ac})/2a$ to find the interval for a polynomial $ax^2 + bx + c$ when $c = 0$.

With these improvements to the to_interval() function, we can appropriately filter the comparisons. In some simple cases, the cost of the filtering process is higher than the cost of the exact computation, so the filtering will not be performed. More precisely, there are three comparison functions:

- compare(Root_of_2, RT); In practice, filtering this function gives no gain, and often adds some random overhead, so, we don’t filter it.
- compare(Root_of_2, FT);
- compare(Root_of_2, Root_of_2).

Experiments showed that in these two cases, when the is_rational flag of the Root_of_2 operands is set to true (meaning that the numbers are rational), filtering compare() in fact slows down the computation, sometimes by more than 10%. Therefore, we chose to filter only comparisons with at least one non-rational operand.

4.4 Reference Counting

After the previous improvements, we profiled the circular kernel using PROF10 and VALGRIND11 to find whether any bottleneck existed or not. We discovered that around 15% of the whole running time was spent on calling the CGAL::MP_Float copy constructor.

This can be explained as follows: The traits class wrapping the circular kernel objects Circular_arc_2 and Line_arc_2 with a boost::variant (Section 5) makes copies of these objects, calling the copy constructor of Circular_arc_point_2, which copies in turn the coordinates (of type Root_of_2), which finally results in copying the three ring type CGAL::MP_Float coefficients of each associated polynomial. Moreover, some predicates and constructions make copies of their own objects as well. Since CGAL::MP_Float is not a basic type but a multi-precision type, its copies and constructions are not trivial, hence not necessarily fast.

A good option to handle this copy construction bottleneck is to use a reference counting12 technique. It consists of storing a handle (which can be seen as a pointer) to a given object, instead of the object itself. When copying an existing object, instead of creating an identical object, we just copy its reference, which is faster in general. For MP_Float, as it is a complex object, the gain is worthy.

The CGAL library offers a reference counting mechanism13. Many functionalities (like compare_x) can benefit from the quick reference equality test (when two objects have the same reference). In order to hide this mechanism from the final user, we provide two different versions of the circular kernel: Simple_circular_kernel_2, without reference counting, and Circular_kernel_2, with reference counting. The user can just choose the kernel that is the most suited to his need. For the interface with the CGAL::Arrangement_2 package, we plug the reference counted kernel.

11 http://valgrind.kde.org/
12 See http://www.memorymanagement.org/articles/recycle.html for an easy to read description.
13 with the CGAL::Handle_for class
<table>
<thead>
<tr>
<th>Input</th>
<th>improved circular kernel (Section)</th>
<th>Default_traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlsi_1</td>
<td>8.42</td>
<td>8.55</td>
</tr>
<tr>
<td>vlsi_2</td>
<td>3.01</td>
<td>2.59</td>
</tr>
<tr>
<td>vlsi_3</td>
<td>25.8</td>
<td>26.7</td>
</tr>
<tr>
<td>vlsi_4</td>
<td>33.4</td>
<td>26.9</td>
</tr>
<tr>
<td>vlsi_5</td>
<td>15.2</td>
<td>14.3</td>
</tr>
<tr>
<td>vlsi_6</td>
<td>135</td>
<td>137</td>
</tr>
<tr>
<td>vlsi_7</td>
<td>185</td>
<td>192</td>
</tr>
<tr>
<td>vlsi_8</td>
<td>445</td>
<td>220</td>
</tr>
<tr>
<td>vlsi_9</td>
<td>525</td>
<td>581</td>
</tr>
<tr>
<td>Very dense</td>
<td>84.5</td>
<td>77.9</td>
</tr>
<tr>
<td>Sparse</td>
<td>0.40</td>
<td>0.51</td>
</tr>
</tbody>
</table>

Table 2: Time, in seconds, spent to compute the arrangement with the improved circular kernel and the Default_traits respectively.

4.5 Benchmarks

Table 2 shows the impact of the improvements described in the previous sections. The enhanced circular kernel is way better than the CGAL 3.2 version (see Table 1). It is even often better than Default_traits. The next section presents further improvements.

5 Final Choices

This section shows the last two steps of the work. The first one consists in changing the representation of algebraic numbers to better control the length of the numbers that are manipulated. Then, geometric filtering completes the enhancement.

5.1 Different Algebraic Number Type Representation

In spite of the overall good performance obtained with the previous improvements (Table 2), the high execution times obtained on the vlsi_8 data set is a show-stopper for the use of the circular kernel on industrial data, since one of the signs of quality should be a reasonably small variation over the data sets of similar size and complexity. This motivated further investigation.

A number of type Root_of_2 is a root of a polynomial $ax^2 + bx + c$ and is represented by the three coefficients $a, b, c$ of a ring type RT and a boolean (see Section 2.2). The basic computations on such an algebraic number reduce to manipulations of the coefficients. When RT is a multi-precision number type (which is necessary to achieve exact computations), the time complexity of these manipulations grows with the length of the numbers. The length increases when computations are cascaded. Moreover normalizing the coefficients by multiplying them by their denominator (see Section 2.2) increases further the length (and performs three additional multiplications).

The to_interval() function is crucial for the filtering process proposed in Section 2.3. Indeed, profiling on the circular kernel shows that approximately 40% of the total running time is spent in the to_interval() function. Most of the time is spent in computing the $\frac{(-b \pm \sqrt{b^2 - 4ac})}{2a}$ expressions.

The vlsi_8 data set generates long numbers (many numbers have up to more than 200 digits, see below), which explains that it heavily suffers from these problems. Different representations of Root_of_2 can be tested to solve them.

- the same representation $a, b, c$, but instead of coefficients in a ring type RT, we use a field type FT. This representation avoids the normalization of the coefficients. However it makes the to_interval() function slower since producing an FT approximation requires one additional division.
- alternatively, we can store the four numbers $A, B, C, D$ of type RT, where $(A + B\sqrt{C})/D$ is the root of $ax^2 + bx + c$ we want to represent.

This representation induces a faster to_interval() function, since it needs less multiplications. But its coefficients still need to be normalized and still grow bigger with cascaded operations, as we can see in the
following example: after computing \(((1 + 2\sqrt{3})/4 + 1/2) + 1/3 + ... + 1/100)\); the values of \(A, B, \) and \(D\) will be respectively \(100!\), \(2 \times 100!\) and \(4 \times 100!\).

- a similar representation \(\alpha, \beta, \gamma\), such that \(\alpha + \beta\sqrt{7}\) is the root, but as numbers of type FT.

This representation suits well our needs. The problem with the sizes of the numbers is partially solved, because only \(\alpha\) grows bigger with addition and subtraction and only two numbers grow with multiplication and division. The coefficients do not need to be normalized, since they are from a field type FT. And though the \texttt{to_interval()} function is slower than with the previous representation, it is still reasonable.

The following pictures present the histograms of the lengths of the numbers that are manipulated through the computation of the arrangement. The abscissa shows the lengths as the number of digits, the ordinate shows the quantity of numbers for each length.

**First representation of Root_of_2** (Section 4):

data set \textit{vlsi\_7}

- data set \textit{vlsi\_8} generates many long numbers

**New representation of Root_of_2** (Section 5.1):

- data set \textit{vlsi\_7}

- data set \textit{vlsi\_8}, there are much fewer large numbers than with the first representation

Table 3 compares the performance of the last representation with the performance of the initial representation. We chose to keep this new representation and adapt the whole code of the algebraic kernel to it. Indeed, \footnote{It is not worth to divide the coefficients in each operation by its \texttt{gcd} (greatest common divisor) since the \texttt{gcd} computation is very costly with arbitrarily big numbers.}

INRIA
cases similar to vlsi_8 are very likely to occur in practice and must be handled in a reasonable running time. The next section shows that the new representation gives the best average running time when coupled with geometric filtering. Moreover, the new representation allows to easily perform more operations on the algebraic numbers, which are necessary when extending this work to dimension 3; this aspect is not elaborated further here.

### 5.2 Geometric Filtering

We mentioned interval arithmetic filtering techniques in Section 4.3.2. A similar scheme can be applied at the geometric level [21] for filtering predicates: fast approximate computation is done first; most of the time, the error bounds allow to certify the result; in bad cases, the filter fails and the result is computed exactly. More precisely, the filtering has three steps:

- **Construction step.** For each object, a shape enclosing it is constructed, with a non-exact type coordinates, e.g., `double`, on which computations are fast.
- **Approximate computation.** The predicate is computed on the enclosing shapes, together with a certificate. When the certificate guarantees that the result is correct, it is returned. Otherwise the next step is performed.
- **Filter failure solving.** The predicate is computed on the exact objects.

There are three conditions required for the effectiveness of this technique. First, the running time of the construction step should be very low. Second, the time taken to run the predicate on the enclosing shapes must be very fast. Third, the filter must be as precise as possible, i.e. the number of filter failures should be as low as possible, otherwise the whole process is just adding an undesirable overhead on the predicate computation. Usually there is a trade-off between the first condition and the precision of the filter.

Using axis-aligned bounding boxes as enclosing shapes is very appropriate for our application (more complicated shapes were tested, without strong evidence that they could perform well [21]). First, such a bounding box is very fast to construct, since it needs only a few `to_interval()` operations, which are already optimized. Second, its storage requires only four `double` coordinates (lower left and upper right corners). Third, predicates like testing whether two bounding boxes intersect are very fast in general. We will see that though bounding boxes approximate circular arcs quite loosely, the precision of the filter is good enough to give very interesting results.

The software design follows again the genericity paradigm: the filtered kernel `Bounding_box_kernel<CircularKernel>` (referred to as `Bounding_box_kernel` in the sequel) is templated by an exact circular kernel.

- **Bounding_box_kernel** redefines the types: for instance, `Circular_arc_with_bbox_2` contains a pointer to a bounding box, that is stored at its first computation, and never recomputed.
- **Bounding_box_kernel** also redefines each predicate by giving a filtered version using the redefined types: when calling a predicate of `Bounding_box_kernel` on circular arcs, the predicate will be first evaluated on the bounding boxes stored in the `Circular_arc_with_bbox_2`, and whenever a filter failure occurs, it is evaluated on the exact objects of type `Circular_arc_2` provided by the exact circular kernel.

Two experiments were conducted to evaluate the performance of this filtering scheme. The first one is done on synthetic data, to measure the impact of the density of the data on the efficiency of the filter. The second experiment is done on the VLSI data sets.
The results confirm our choice for the new representation: even though the representation of Section 4 obtains slightly better running times on many cases, the very good results achieved on vlsi_8 show that the representation of Section 5.1 has a more regular behavior and in fact the best average running time. The performance is also better in all cases than the default traits class used as reference.

In this experiment, we consider a scenario with 100 uniformly random circles with unit radius and center in a square with side length $L$, varying from near 0 to 1000. We compute the arrangement of these circles as in previous experiments (with the hardware configuration presented in Section 3), with both the filtered circular kernel and the non filtered circular kernel.

The figure plots the ratio between the two running times, as a function of the size length $L$ of the square. When $L$ is very small, the filtered kernel tends to have the same performance as the non filtered kernel, since the number of intersections between the circles is big. As $L$ gets bigger, the filtered version of the circular kernel tends to be twice as fast as the non filtered.

The following table shows the impact of the bounding box filtering on the circular kernel for industrial data sets, both with the first representation of the Root_of_2 numbers (improved in Section 4) and the new representation (Section 5.1).

<table>
<thead>
<tr>
<th>Input</th>
<th>b. box, Sec. 4</th>
<th>b. box, Sec. 5.1</th>
<th>Default_traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlsi_1</td>
<td>4.96</td>
<td>4.61</td>
<td>8.35</td>
</tr>
<tr>
<td>vlsi_2</td>
<td>1.01</td>
<td>1.31</td>
<td>2.59</td>
</tr>
<tr>
<td>vlsi_3</td>
<td>17.7</td>
<td>21.8</td>
<td>26.7</td>
</tr>
<tr>
<td>vlsi_4</td>
<td>27.5</td>
<td>25.4</td>
<td>26.9</td>
</tr>
<tr>
<td>vlsi_5</td>
<td>14.2</td>
<td>14.8</td>
<td>14.3</td>
</tr>
<tr>
<td>vlsi_6</td>
<td>106</td>
<td>134</td>
<td>137</td>
</tr>
<tr>
<td>vlsi_7</td>
<td>152</td>
<td>169</td>
<td>192</td>
</tr>
<tr>
<td>vlsi_8</td>
<td>332</td>
<td>136</td>
<td>220</td>
</tr>
<tr>
<td>vlsi_9</td>
<td>437</td>
<td>492</td>
<td>581</td>
</tr>
<tr>
<td>Very dense</td>
<td>81.0</td>
<td>76.2</td>
<td>77.9</td>
</tr>
<tr>
<td>Sparse</td>
<td>0.25</td>
<td>0.21</td>
<td>0.51</td>
</tr>
</tbody>
</table>

In the figure on the right side, the first experiment was repeated, showing now the ratio between the performance of the **Bounding_box_kernel** (on the new representation of Sec. 5) and the **Default_traits**. As we can see, when the scenario is very dense the **Default_traits** can be as much as 20% faster than the **Bounding_box_kernel**, since filtering induces an overhead and often fails. However, as $L$ grows, the **Bounding_box_kernel** takes very soon the advantage, stabilizing at three time faster than the **Default_traits**.

### Conclusion and future work

We showed how several techniques can be used to combine generality and re-usability of the functionality, together with robustness and efficiency. This work is currently being generalized to the more complicated case of manipulations of spheres, circles and circular arcs in 3D.

It should be investigated whether the promising recent framework for filtering constructions [16] can be adapted to the case of curved objects.

**Acknowledgments.** We thank MANIA BARCO and Andreas Fabri from GEOMETRYFACTORY for giving us access to industrial VLSI data.
References


