**ACS**

Algorithms for Complex Shapes
with Certified Numerics and Topology

# An implementation for the 2D Algebraic Kernel

Pavel Emeliyanenko Michael Kerber

ACS Technical Report No.: ACS-TR-363602-01

| | |
|---|---|
| Part of deliverable: | WP-III/D6 |
| Site: | MPI |
| Month: | 36 |

Project co-funded by the European Commission within FP6 (2002–2006)
under contract nr. IST-006413

**Abstract**

We report on a model for CGAL's AlgebraicKernelWithAnalysis_d_2 concept which refines AlgebraicKernel_d_2. Our implementation handles bivariate polynomials in full generality, i.e., with no restriction on their degree. Moreover, it allows both integers and nested square-root numbers as coefficient type. The Curve analysis and Curve pair analysis required by the concept are realized using recent work of Eigenwillig, Kerber and Wolpert ('Fast and exact geometric analysis…', ISSAC 2007 and 'Exact and efficient 2D-Arrangements…'. SODA 2008). The consequent use of certified numerical methods leads to significant speed-ups without spoiling exactness. We present benchmark results about the performance of several key methods.

## 1 Introduction

The CGAL concepts AlgebraicKernel_d_1 and AlgebraicKernel_d_2 encapsulate basic algebraic operations for uni- and bivariate polynomials. Models for the univariate kernel have been already provided, compare [11]. In this report, we introduce an exact, complete, efficient and generic model for the two-dimensional case. Within the ACS-project, a 2D kernel for circular arcs [17] (extended in [12]), and an experimental kernel for the general case, based on SYNAPS [16], have been presented. The most challenging operations required by the concept are

- Solve_2: Computing the real solutions of a system of two bivariate polynomials
- SignAt_2: Computing the sign of a polynomial at some point defined by algebraic coordinates
- CompareXY_2: Comparing points lexicographically
- RefineX_2 / RefineY_2: Computing numerical approximations of algebraic coordinates

More precisely, our implementation constitutes a model of CGAL's AlgebraicKernelWithAnalysis_d_2 concept which refines AlgebraicKernel_d_2. This extends the kernel by functionality to analyze a single curve, or a pair of curves [3]. Each analysis constitutes a substantial research result in its own, we refer to the original articles about Curve analysis [9] and Curve pair analysis [8]. The general strategy of our algebraic kernel is to use these analyses to implement all methods required by the concept.

Efficiency is obtained in both analysis steps by combining symbolic methods, such as subresultant computations and (modular) gcd computation, with numerical filter techniques, and certified numerical methods, such as the Bitstream Descartes method [10], [7], which computes the real roots of a univariate polynomial only by approximations of the coefficients. Furthermore, we achieve speed-ups due to caching, and by using a lazy-evaluation strategy, i.e., analysis steps are only performed when they are needed to answer some query.

Our implementation allows to exchange the underlying coefficient type of the polynomials easily by a template parameter: we tested two coefficient types that model the integers, but also nested square root expressions are allowed as coefficient type using CGAL's new Sqrt_extensions.

The presented model has been used already for implementations of several geometric algorithm, such as the arrangement computation of algebraic curves in the plane [8] and on a torus [4], and the analysis of algebraic surfaces of any degree [5]. We report on some benchmarks of our approach that covers major parts of the implementation.

## 2 Curve and curve pair analysis

The fundament of our implementation for the algebraic kernel is formed by two main operations: first, given a bivariate polynomial $f$, perform a *curve analysis* for the algebraic curve with defining equation $f = 0$. That means roughly, compute the topology of the curve, plus the position of critical points of the curve. Second, given two algebraic curves, perform a *curve pair analysis* which basically means to compute their intersection points. The AlciX library of Exacus provides curve and curve pair analysis for algebraic curves of arbitrary degree and bitsize. For that, AlciX offers two template classes as an interface for the results of the curve and curve pair analysis:

```
template < typename ArithmeticTraits, typename Coefficient >
    Algebraic_curve_2;

template < typename AlgebraicCurve_2 >
    Algebraic_curve_pair_2;
```

A model of Exacus' ArithmeticTraits defines a coherent collection of internally used number types (integers, rationals, polynomials) [1]. By default, the coefficient type is defined to be the integer type of the first template argument, but also other types can be chosen (e.g., an instance of `CGAL::Sqrt_extension` that is inter-operable with the types of the ArithmeticTraits).

These classes model the AlgebraicCurve_2 and AlgebraicCurvePair_2 concepts from Exacus for arbitrary degree. They have been introduced as a generic interface to come up with arrangement computations of algebraic curves, and models for conic and cubic curves have been provided already. We refer to [2] for an overview. We next describe the most important functions of both classes:

**Algebraic_curve_2:** An instance is initialized using the defining bivariate polynomial, and stores a set of `Vert_line` objects. Each such `Vert_line` contains information about the curve at some $x$-coordinate:

`x()` $x$-coordinate of the `Vert_line`, in isolating interval representation.

`num_arcs()` Number of fiber points of the curve at this $x$-coordinate

[1]This concept is also available in CGAL as ArithmeticKernel

`lower_boundary(i)`, `upper_boundary(i)` Isolating interval for the $i$th fiber point

`refine(i)` Refines the $i$th isolating interval

`num_arcs_left(i)`, `num_arcs_right(i)` Number of arcs that approach the $i$th fiber point from the left and form the right

`num_arcs_approaching_vertical_asympotote`
`(minus_left, minus_right, plus_left, plus_right)` Number of arcs that have a vertical asymptote at this x-coordinate

`has_vertical_line()` Denotes whether the curve has a vertical line at this x-coordinate as component

The `Algebraic_curve_2` instance has the following main functions:

`event_info_at_x(x_0)` Return a `Vert_line` object at $x$-coordinate `x_0`

`num_events()` Number of real resultant roots of the curve (see explanation below)

`event_info(i)` Return a `Vert_line` object at the $i$th resultant root

For the computation, the following strategy is used: when the first `Vert_line` object is queried, the $x$-coordinate of all critical points and of vertical asymptotes is computed by isolating the real roots of the resultant polynomial $\mathrm{res}_y(f, \frac{\partial f}{\partial y})$. The `Vert_line` objects are only computed on demand, i.e., the computation is delayed until the `Vert_line` is first queried by the user, and cached afterwards to prevent multiple calculation.

The technique to compute the geometric data is described in detail in [9].

**Algebraic_curve_pair_2:** A curve pair is initialized using two instances of `Algebraic_curve_2`. It stores a set of `Event2_slice` objects which basically contains, for some $x$-coordinate, the vertical ordering of curve arcs of the first curve, curve arcs of the second curve, and intersection points. For that, one can ask for a `Arc_at_event_2_container`, or a `Arc_at_interval_2_container`, that contains a vector of tokens `CURVE1, CURVE2` or `CURVE_BOTH` to reflect the ordering.

The interface of `Algebraic_curve_pair_2` contains the following methods:

`curve1()`, `curve2()` Accesses the `Algebraic_curve_2` instances of the curve pair.

`event_x(i)` The $x$-coordinate of the $i$th event of the curve pair (i.e., a resultant root of a single curve, or of the curve pair)

`slice_at_event(i)` Returns a `Event2_slice` object for the $i$th event

`slice_at_interval(i)` Returns a `Event2_slice` object for the interval between the $(i-1)$st and $i$th event

The computation scheme is similar as for a single curve: when the first slice or the first $x$-coordinate is queried, the critical $x$-coordinates are computed with root isolation of the resultant polynomial $\text{res}_y(f,g)$ (where $f$ and $g$ are the equations of the curves). Again, only slices that are asked for are actually computed, and then cached. This brings computational advantages, as not all slices need to be computed in typical applications. For details how to obtain the slices, we refer to [8].

## 3 Interface of the kernel

We introduce a model for the AlgebraicKernelWithAnalysis_d_2 concept of CGAL. It is located in the package `Algebraic_kernel_d` in the current CGAL trunk. Its interface looks as follows:

```
template < typename AlgebraicCurvePair_2, typename AlgebraicKernel_1 >
   class Algebraic_curve_kernel_2
```

AlgebraicCurvePair_2 must be instantiated with a model of the corresponding EXACUS concept which has been specified in the previous section. Internally, the univariate kernel defined by AlgebraicKernel_1 is used.

The kernel uses the $x$-coordinate representation defined in the Algebraic-CurvePair_2 model (in our case, this is the *isolating interval representation* as defining polynomial plus isolating interval). To represent elements $(\alpha,\beta) \in \mathbb{R}^2$, the class `XY_coordinate` is used: it contains a triple `(alpha,curve,arcno)`, where `alpha` is a representation of the $x$-coordinate $\alpha$, `curve` is a curve on which $(\alpha,\beta)$ lies, and `arcno` denotes the index of the point $(\alpha,\beta)$ in the fiber of `curve` over `alpha`. Its main functions are the following

`x(), curve(), arcno()` Accesses the data members

`compare_xy(b)` Lexicographically compares with `b`

`get_approximation_x(), get_approximation_y()` Approximates the coordinates with a rational interval

`refine_x(), refine_y()` Refines the approximations

`y()` Obtains an isolating interval representation of the $y$-coordinate

To satisfy the AlgebraicKernel_2 concept, the `Algebraic_curve_kernel_2` has the following functors

`IsSquareFree_2, IsCoprime_2` Checks for squarefreeness of a polynomial, and coprimality of a curve pair

`MakeSquareFree_2, MakeCoprime_2` Make a polynomial square free, or two polynomials coprime

`SquareFreeFactorize_2` Decomposes a polynomial into its square free components

`Solve_2` Computes the real solutions of systems of two bivariate polynomials

`XCriticalPoints_2`, `YCriticalPoints_2` Computes all $x$- or $y$-critical points of a curve

`Sign_At_2` Evaluates the sign of a polynomial for a given `XY_coordinate`

Additionally, there are several functors that simply forward to the corresponding member functions of `XY_coordinate` (`GetX_2`, `RefineX_2`, etc.). See the documentation of the concept [3] for a complete list.

`Algebraic_curve_kernel_2` also satisfies the AlgebraicKernelWithAnalysis_d_2 concept. For that, it defines the two nested types `Curve_analysis_2` and `Curve_pair_analysis_2` that provide methods to analyze single curves, and curve pairs. Both types have their own nested type `::Status_line_1` for geometric information at a certain $x$-coordinate. The data and member functions stored in those objects is, up to renaming, equal to those that are provided by AlciX `Algebraic_curve_2` and `Algebraic_curve_pair_2` classes. Therefore, we skip a more detailed description of the interface.

`Algebraic_curve_kernel_2` also has the two functors `Construct_curve_2` and `Construct_curve_pair_2` to obtain a curve (pair) analysis. Internally, the effect is that the objects are cached and multiple expensive computations are prevented.

## 4 Implementation of kernel functions

### 4.1 Algebraic_curve_kernel_2

Most of the kernel functors are implemented by exploiting the information of the curve and curve pair analysis. In that sense, `Algebraic_curve_kernel_2` works simply as adaptor for the corresponding AlciX functions. We describe the proceeding for most of the functors next:

**`CompareXY_2`:** Lexicographic comparison of two `XY_coordinate_2`, let them be represented by `(alpha1,curve1,arcno1)` and `(alpha2,curve2,arcno2)`: we compare their $x$-coordinates `alpha1` and `alpha2`, using the comparison function from the univariate algebraic kernel that is passed as template parameter. If `alpha1!=alpha2`, nothing remains to do. If they are equal, and `curve1==curve2`, we simply compare the arc numbers for the result. Otherwise, we construct the curve pair induced by `curve1` and `curve2`, and construct the `Event2_slice` at `alpha1`. Using this slice, one can read off whether the (`arcno1`)th arc of `curve1` is above or below the (`arcno2`)th arc of `curve2`, or whether they are intersecting.

**`Solve_2`:** Find common solutions of $f, g \in \mathbb{R}[x, y]$: Build the curve pair analysis of $f$ and $g$, iterate through all the event coordinates, and find the intersection points in the stacks. Return all intersection points.

**`XCriticalPoints_2`, `YCriticalPoints_2`:** Apply `Solve_2` on $f$ and its derivative with respect to $x$ or $y$.

`SignAt_2`: Assume we want to compute the sign of $f \in \mathbb{R}[x,y]$ at $(\alpha,\beta)$, represented as (`alpha,curve,arcno`). For that, compute the the curve pair analysis of $f$ and `curve`, and compute the slice at `alpha`. Check whether the (`arcno`)th arc of `curve` is an intersection. If yes, the sign of $f(\alpha,\beta)$ is zero.

Otherwise, the sign is not zero. Obtain an approximation of $(\alpha,\beta)$ using `XY_coordinate::get_approximation_x()` and `::get_approximation_y()`, and evaluate $f$ at this approximation, using interval arithmetic. If the resulting interval does not contain zero, the sign is determined. If it contains zero, refine the approximation using `XY_coordinate::refine_x()` and `XY_coordinate::refine_y()`, and repeat.

`CurveAnalysis_2`, `CurvePairAnalysis_2`: These classes simply form an adapter to the AlgebraicCurve_2 and AlgebraicCurvePair_2 concepts from Exacus.

`SquareFreeFactorize_2`: The algorithm to decompose a polynomial in its square free components is available as part of Cgal's `Polynomial_traits_d` template class.

The decomposition is done via iterated gcd computations, which are performed with a modular algorithm (compare [13])

`RefineX_2`: Generically calls the refinement method of the univariate kernel. Our implementation uses the quadratic interval refinement method [1].

`RefineY_2`: Generically calls the refinement method of the associated `Vert_line` object. In our implementation, this is done by continuing the Bitstream Descartes method on the isolating interval, until an isolating interval of smaller size is found.

`GetY_2`: Given some `XY_coordinate_2` object `(alpha, curve, arcno)` that represents $(\alpha,\beta)$, the task is to compute an isolating interval representation for $\beta$. This is a non-trivial task, since one has to find a polynomial in the suitable coefficient space that has $\beta$ as one of its root. Assume `curve` has the defining equation $f \in \mathbb{Z}[x,y]$, the polynomial $f(\alpha,y)$ has the $\beta$ as root, but its coefficients are algebraic in general, and not integers.

To find the polynomial, we proceed as follows: first, we check how many arcs of `curve` are leaving $(\alpha,\beta)$ to the left and to the right, using the `Vert_line` object. Let $(l,r)$ denote these numbers. If $(l,r) \neq (1,1)$, the point is known to be $y$-critical, and thus, its $y$-coordinate is a root of $R := \mathrm{res}_x(f, \frac{\partial f}{\partial y})$, where $f$ is the defining equation for `curve`. We take $R$ as defining equation for $\beta$.

Otherwise, we exploit that $\beta$ is intersection point of `curve` and the vertical line curve $x = \alpha$.[2] The equation $g$ of this vertical curve is known by the representation of $\alpha$. So, $R := \mathrm{res}_x(f,g)$ has $\beta$ as a root, and we used this as defining equation.

[2]we assume for simplicity that `curve` does not contain the whole vertical line

We remark that the first part is used as a filter to prevent the computation of $\text{res}_x(f, g)$ if possible. The reason is that typically, $g$ is itself a resultant polynomial of quite high degree and so, $R$ becomes quite complex.

Having a defining equation for $\beta$, we need to find an isolating interval. For that, isolate the real roots of the defining equation, and by refining the approximation of $\beta$ (using `XY_coordinate::refine_y()`), determine which of the intervals represents $\beta$.

`CompareY_2`: We want to compare the $y$-coordinates of `(alpha1,curve1,arcno1)` and `(alpha2,curve2,arcno2)`, If `alpha1==alpha2` (checked by `CompareX`), we call `CompareXY` to get the result, since the lexicographical ordering equals the $y$-ordering in this case. Otherwise, we call `GetY` on both representations to obtain isolating interval representations of both $y$-coordinates, and compare them using `CompareX`.

We recommend to use the methods `GetY` and `CompareY_2` carefully, because the representation of $y$-coordinates in isolating interval representation does not really fit to our projection-based model. Thus, these methods causes a serious overhead of symbolic computation. The user should check whether such a representation is really necessary for his application. For instance, computing arrangements of arbitrary algebraic curves is possible without the use of these functions.

We remark that our approach for `Solve_2` is clearly not optimal for the problem when considered in isolation: For finding the intersection points of two curves, it is certainly not the best solution to analyze each single curve first and perform a curve pair analysis afterwards. However, the additional cost of performing a curve analysis for each occurring curve might amortize when a curve is intersected with many other curves. This is particularly the case when arrangements of algebraic curves are computed.

## 4.2 Filtered_algebraic_curve_kernel_2

Performing a curve analysis and a curve pair analysis are expensive operations, mainly because the trigger a symbolic (sub)resultant computation. Techniques to prevent the computation of at least some such analysis objects help to improve the performance in applications. We therefore also implemented a filtered kernel:

```
template < typename AlgebraicCurvePair_2,
           typename AlgebraicKernel_1 >
    class Filtered_algebraic_curve_kernel_2;
```

It redefines the predicates `CompareXY_2`, `CompareY_2` and `SignAt_2`. In all cases, the `XY_coordinate` objects are first approximated up to a certain *threshold precision*, and the predicated is tried to be answered using only this approximation. If this fails, the usual predicated is called.

An application of this kernel is discussed in another ACS-report [15].

## 5 Experiments

Our benchmark program, called `Algebraic_curve_kernel_2` is available in the CGAL trunk in the benchmark folder of the `Algebraic_kernel_d` package. As input, it gets a file specifying bivariate polynomials (currently, those polynomials have to be provided in EXACUS format). The program than performs three steps:

1. Compute all pairwise intersection points, and store the `Xy_coordinate_2` objects in a set $S$.
2. Sort the elements of $S$ lexicographically
3. Refine each point to double precision (53 bits)

The crucial operations tested in each step are `Solve_2`, `CompareXY_2`, and `RefineX_2, RefineY_2`, respectively. We remark that an analogue benchmark program was used for MPI's univariate kernel [14].

Our experiments were performed on an AMD Dual-Core Opteron(tm) 8218 (1 GHz) multi-processor platform with a total memory of 32 GB, running Debian Etch. The program was compiled using gcc-4.1.2 with compiler flags `-O2` and `-DNDEBUG`. The internal CGAL release 3.4-I-270 has been used for the benchmarks. This release was configured using boost-1.33.1, gmp-4.2 and mpfr-2.2.0. Also, the program requires the ALCIX library plus other parts of EXACUS. A tarball with the files used is available on demand. ALCIX internally requires the NTL-library (version 5.4).

For all tests, we used the `Filtered_algebraic_curve_kernel_2`, described in Section 4.2. To investigate the implementation's behavior for the generic case (i.e., without degeneracies), we first look at randomly generated curves. We only consider dense polynomials in the tests, three parameter are varied in the benchmarks: the total degree of the polynomials, the maximal bitsize per coefficient, and the number of polynomials considered. For all tests, we made 5 independent test runs and took the median with respect to the total running time.

First, we consider polynomials of increasing degrees, with bitsize up to 50 per coefficient, and 10 polynomials per test instance (see Table 1).

We see that with increasing degree, the `Solve_2` functor becomes more dominant in the overall running time. This is not surprising since it includes all curve and curve pair analyzes of the curves and thus contains all the algebraic operations of the algorithm. Also, the `to_double` operation becomes more costly. The reason is that the solutions of the systems have a higher-degree representation what makes their refinement more expensive. Compared to these operations, the time for sorting is negligible.

Second, we fix the degree to 5, and vary the bitsize, again with 10 polynomials per instance (Table 2).

We can see that with increasing bitsize, the cost for solving also increases, which is caused by the higher bit-complexity. Also, the sorting becomes slightly more expensive, due to the higher costs of the modular gcd algorithm. However,

Table 1: Random curve, varying degree

| deg | # points | Solve | Sort | to_double | total |
|---|---|---|---|---|---|
| 3 | 99 | 0.24 | 0.02 | 0.41 | 0.67 |
| 5 | 147 | 0.73 | 0.05 | 1.36 | 2.14 |
| 7 | 195 | 2.33 | 0.15 | 3.12 | 5.60 |
| 9 | 269 | 9.52 | 0.33 | 18.68 | 28.53 |
| 11 | 249 | 33.01 | 0.46 | 32.48 | 65.95 |
| 13 | 213 | 103.91 | 0.60 | 41.32 | 145.83 |
| 15 | 229 | 297.46 | 0.80 | 87.76 | 386.02 |

Table 2: Random curves, varying bitsize

| bitsize | # points | Solve | Sort | to_double | total |
|---|---|---|---|---|---|
| 25 | 187 | 0.65 | 0.07 | 1.84 | 2.56 |
| 50 | 121 | 0.70 | 0.04 | 1.18 | 1.92 |
| 100 | 127 | 0.82 | 0.04 | 1.21 | 2.07 |
| 200 | 141 | 1.19 | 0.05 | 1.21 | 2.45 |
| 400 | 119 | 2.21 | 0.05 | 0.99 | 3.25 |
| 800 | 143 | 5.72 | 0.06 | 1.17 | 6.95 |
| 1600 | 137 | 15.13 | 0.07 | 1.20 | 16.40 |
| 3200 | 181 | 41.41 | 0.13 | 1.48 | 43.02 |
| 6400 | 129 | 116.83 | 0.15 | 1.15 | 118.13 |

the bitsize has practically no effect on the `to_double` operation (for that, the number of solution is the crucial quantity). We take this as an advantage of our implementation of the refinement method, which approximates the coefficients and interval boundaries by bigfloat numbers and works with the bigfloats instead of the exact representation.

Finally, we fix degree 5 and bitsize 50, and vary the number of input polynomials (Table 3).

We see that in this setting, the `Solve_2` functor is not dominant anymore. The reason is of course that there are more intersection points to be considered, and so the sorting and approximating becomes more expensive in total. Note that the cost for solving is basically proportional to the number of curve pair analyzes. Also, the cost for approximation is proportional to the number of intersection points, but the cost of sorting grows superlinearly with respect to them. The reason is that for many intersection curves, we expect more pairs that are close to each other, so their comparison cannot be computed by our filters, and exact methods have to be applied.

To investigate the behavior for degenerate situations, we randomly generate trivariate polynomials $f_1, \ldots, f_n$ and take all pairwise resultants $\mathrm{res}_z(f_i, f_j)$, plus all silhouette curves $\mathrm{res}_z(f_i, \frac{\partial f_i}{\partial z})$ as input. It is well-known that such curves

Table 3: Random curves, varying number of polynomials

| # pols | # points | Solve | Sort | to_double | total |
|---|---|---|---|---|---|
| 10 | 147 | 0.75 | 0.05 | 1.46 | 2.26 |
| 25 | 938 | 4.21 | 0.50 | 9.19 | 13.90 |
| 40 | 2500 | 10.43 | 1.67 | 22.76 | 34.86 |
| 55 | 4827 | 19.89 | 3.62 | 47.78 | 71.29 |
| 70 | 7201 | 32.54 | 5.78 | 68.30 | 106.62 |
| 85 | 10638 | 45.79 | 8.82 | 102.75 | 157.36 |
| 100 | 14722 | 65.01 | 12.84 | 139.06 | 216.91 |

produces in general singular points as well as tangential intersections and are therefore well-suited for systematic benchmarks.

In the first setting, we choose 3 surfaces with bitsize 10 per coefficient, and varying degree $n$. This produces 3 projected silhouettes curves of degree up to $n(n-1)$, and 3 projected intersection curves of degree $n^2$.

Table 4: Projection curves of surfaces, varying degree

| degree | # points | Solve | Sort | to_double | total |
|---|---|---|---|---|---|
| 2 | 52 | 0.30 | 0.02 | 0.20 | 0.52 |
| 3 | 185 | 3.32 | 0.15 | 4.07 | 7.54 |
| 4 | 160 | 146.28 | 0.68 | 45.33 | 192.29 |
| 5 | 149 | 6172.63 | 3.56 | 1028.74 | 7204.93 |

Second, we choose 3 surfaces of degree 3 with varying bitsize. The second column denotes the mean bitsize of the occurring curves.

Table 5: Projection curves of surfaces, varying bitsizes

| degree | curve bitsize | # points | Solve | Sort | to_double | total |
|---|---|---|---|---|---|---|
| 10 | 48.83 | 185 | 3.30 | 0.14 | 4.12 | 7.56 |
| 20 | 99.16 | 133 | 4.25 | 0.23 | 2.38 | 6.86 |
| 40 | 197.33 | 59 | 9.33 | 0.18 | 0.95 | 10.46 |
| 80 | 399.33 | 97 | 26.07 | 0.69 | 1.12 | 27.99 |
| 160 | 800.33 | 85 | 74.35 | 2.50 | 0.89 | 77.74 |
| 320 | 1600.00 | 81 | 215.66 | 6.68 | 0.86 | 223.20 |
| 640 | 3199.67 | 97 | 624.88 | 24.17 | 1.11 | 649.94 |

Finally, we fix the surface degree to 3, the bitsize to 10 bits, and vary the number of surfaces. Note that the number of curves grows quadratically.

The results show a similar result compared to the generic instances. We conclude that degenerate situations do not affect the characteristics of the imple-

Table 6: Projection curves of surfaces, varying surface number

| # surfaces | # points | Solve | Sort | to_double | total |
|---|---|---|---|---|---|
| 2 | 4 | 0.68 | 0.00 | 0.04 | 0.72 |
| 3 | 99 | 2.25 | 0.09 | 1.39 | 3.73 |
| 4 | 309 | 6.55 | 0.39 | 4.90 | 11.84 |
| 5 | 865 | 18.45 | 1.38 | 35.15 | 54.98 |
| 6 | 1455 | 33.70 | 2.95 | 47.38 | 84.03 |
| 7 | 2652 | 64.01 | 5.61 | 115.26 | 184.88 |
| 8 | 4500 | 109.78 | 10.70 | 166.40 | 286.88 |

mentation too much, although of course, degeneracies (often) lead to additional symbolic computations and worsen the running time.

## 6 About maturation

Some remarks about the history, the current status, and the next steps for this package: the `AlciX` classes `Algebraic_curve_2` and `Algebraic_curve_pair_2` have been implemented by Michael Kerber, starting in February 2006. Most parts of the `Algebraic_curve_kernel_2` class have been written by Pavel Emeliyanenko. The current state of having an experimental Cgal package relying on external libraries is clearly an intermediate state. We are working on the migration of the AlciX library into the `Algebraic_kernel_d` package as part of the merge of Exacus into Cgal. However, as AlciX was initially built on top of the Exacus libraries Support, NumeriX and SweepX, all the functionality of those packages must be merged into Cgal first. This process is now about to be finished, and we will start with the movement of AlciX within the next months.

Although the implementation is superficially in a preliminary state, we still claim that it can be called mature with respect to its methods: the underlying algorithms constitute novel research contributions [9], [8], and considerable efforts have been taken to achieve practical efficiency in sub-algorithm. For instance, a fast method for computing subresultants [6], and the quadratic interval refinement algorithm [1] have been implemented. Also, modular resultant computation using MAPLE (and its OpenMaple interface) is implemented, but we did not use it for our benchmarks to simplify reproducibility of the results. An integration of a modular resultant algorithm into Cgal would certainly improve the algorithm's performance further, especially for generic instances.

## Acknowledgements

We are indebted to the whole Exacus team, especially to Eric Berberich, Arno Eigenwillig and Michael Hemmer for all support on the implementation.

## References

[1] J. Abbott. Quadratic interval refinement for real roots. Poster presented at the 2006 International Symposium on Symbolic and Algebraic Computation (ISSAC 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. of the 13th Annual European Symposium on Algorithms (ESA 2005)*, volume 3669 of *LNCS*, pages 155–166. Springer, 2005.

[3] E. Berberich, M. Hemmer, M. I. Karavelas, and M. Teillaud. Revision of the interface specification of algebraic kernel. Technical Report ACS-TR-243301-01, INRIA Sophia-Antipolis, Max Planck Institut für Informatik, National University of Athens, 2007.

[4] E. Berberich and M. Kerber. Exact arrangements on tori and dupin cyclides. In *Proc. of the 2008 ACM Symp. on Solid and Physical Modeling, (SPM 08)*, 2008. to appear.

[5] E. Berberich, M. Kerber, and M. Sagraloff. Exact geometric-topological analysis of algebraic surfaces. In *Proc. of the 24th Ann. Symp. on Computational Geometry (SCG'08)*, 2008. to appear.

[6] L. Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145:149–163, 2000.

[7] A. Eigenwillig. *Real Root Isolation for Exact and Approximate Polynomials Using Descartes' Rule of Signs*. PhD thesis, Universität des Saarlandes, Germany, 2008.

[8] A. Eigenwillig and M. Kerber. Exact and efficient 2d-arrangements of arbitrary algebraic curves. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA08)*, 2008. 122–131.

[9] A. Eigenwillig, M. Kerber, and N. Wolpert. Fast and exact geometric analysis of real algebraic plane curves. In C. W. Brown, editor, *Proocedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC 2007)*, pages 151–158, 2007.

[10] A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, and N. Wolpert. A Descartes algorithm for polynomials with bit-stream coefficients. In *8th International Workshop on Computer Algebra in Scientific Computing (CASC 2005)*, volume 3718 of *LNCS*, pages 138–149, 2005.

[11] I. Emiris, M. Hemmer, M. Karavelas, M. Kerber, B. Mourrain, E. P. Tsigaridas, and Z. Zafeirakopoulos. Cross-benchmarks of univariate algebraic kernels. Technical Report ACS-TR-363602-02, INRIA and MPI and NUA, 2008.

[12] I. Emiris and E. Tsigaridas. Cgal package for 2d curved kernel (a wrapper for synaps). Technical Report ACS-TR-123203-02, 2006.

[13] M. Hemmer and D. Hülse. Traits classes for polynomial gcd computation over algebraic extensions. Technical Report ACS-TR-241405-03, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut fr Informatik, Saarbrcken, GERMANY, 2007.

[14] M. Hemmer and S. Limbach. Benchmarks on a generic univariate algebraic kernel. Technical Report ACS-TR-243306-03, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[15] M. Kerber. On filter methods in CGAL's 2d curved kernel. Technical Report ACS-TR-243404-03, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2008.

[16] B. Mourrain, J. Pavone, and J. Wintz. Benchmarks and evaluation of an experimental algebraic kernel. Technical Report ACS-TR-243306-04, 2007.

[17] S. Pion and M. Teillaud. Cgal package for 2d curved kernel. Technical Report ACS-TR-123203-01, INRIA, 2006.